The full source code is reproduced below. The archive at the end of this post contains the source code and a compiled executable.

Main.cpp

```
#include <Windows.h>
#include <wchar.h>
#include <stdio.h>
#include "Injector.h"
#include "Encrypter.h"

#define BB(x) __asm _emit x

#define STRING_COMPARE(str1, str2) \
    __asm push str1 \
    __asm call get_string_length \
    __asm push eax \
    __asm push str1 \
    __asm mov eax, str2 \
    __asm push eax \
    __asm call strings_equal

#pragma code_seg(".inject")
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
    }
    __asm { //Decrypt all sections
        push kernel32_image_base
        push target_image_base
        call decrypt_sections
    }
    //Any additional code can go here
    __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
    }

    ////////////////////////////////////////////////////////////////////
    //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
    }
    ////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    //Gets the length of the string passed as the parameter
    //Call as int get_string_length(char *str)
    ////////////////////////////////////////////////////////////////////
    __asm {
    get_string_length:
        push ebp
        mov ebp, esp
        mov edi, [ebp+0x8]      //String held here
        mov eax, 0x0            //EAX holds size of the string
```

```
        counting_loop:
            cmp byte ptr[edi], 0x0//Current byte is null-terminator?
            je string_done      //Done, leave loop
            inc edi             //Go to next character
            inc eax             //size++
            jmp counting_loop
        string_done:
            mov esp, ebp
            pop ebp
            retn
    }
    ////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    //String comparison function, checks for equality of two strings
    //Call as bool strings_equal(char *check_string, char *known_string, int
known_string_length)
    ////////////////////////////////////////////////////////////////////
    __asm {
    strings_equal:
        push ebp
        mov ebp, esp
        mov eax, 0x0            //Assume unequal
        cld                     //Forward comparison
        mov esi, [ebp+0x8]      //ESI gets check_string
        mov edi, [ebp+0xC]      //EDI gets known_string
        mov ecx, [ebp+0x10]     //ECX gets known_string_length
        repe cmpsb              //Start comparing
        jne end
        mov eax, 0x1            //Strings equal
    end:
        mov esp, ebp
        pop ebp
        ret 0xC
    }
    ////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    //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
        }
    ////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    //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
        }

    __asm {
    virtualprotect:
        call virtualprotectback
        BB('V') BB('i') BB('r') BB('t') BB('u') BB('a') BB('l')
        BB('P') BB('r') BB('o') BB('t') BB('e') BB('c') BB('t') BB(0)
    rdata_name:
        call rdata_nameback
        BB('.') BB('r') BB('d') BB('a') BB('t') BB('a') BB(0)
    rsrc_name:
        call rsrc_nameback
        BB('.') BB('r') BB('s') BB('r') BB('c') BB(0)
    section_name:
        call section_nameback
        BB('.') BB('i') BB('n') BB('j') BB('e') BB('c') BB('t') BB(0)
        int 0x3                 //Function signature
        int 0x3
        int 0x3
```

```
        int 0x3
    }
}
#pragma code_seg()
#pragma comment(linker, "/SECTION:.inject,re")

wchar_t *convert_to_unicode(char *str, unsigned int length) {
    wchar_t *wstr;
    int wstr_length = MultiByteToWideChar(CP_ACP, 0, str, (length + 1), NULL,
0);
    wstr = (wchar_t *)malloc(wstr_length * sizeof(wchar_t));
    wmemset(wstr, 0, wstr_length);
    if (wstr == NULL)
        return NULL;
    int written = MultiByteToWideChar(CP_ACP, 0, str, length, wstr,
wstr_length);
    if(written > 0)
        return wstr;
    return NULL;
}

int main(int argc, char* argv[]) {
    if(argc != 2) {
        printf("Usage: ./%s <target>\n", argv[0]);
        return -1;
    }
    wchar_t *target_file_name = convert_to_unicode(argv[1], strlen(argv[1]));
    if(target_file_name == NULL) {
        printf("Could not convert %s to unicode\n", argv[1]);
        return -1;
    }
    pfile_info target_file = file_info_create();
    void (*stub_addr)(void) = injection_stub;
    unsigned int stub_size = get_stub_size(stub_addr);
    unsigned int stub_size_aligned = 0;
    bool map_file_success = map_file(target_file_name, stub_size, false,
target_file);
    if(map_file_success == false) {
        wprintf(L"Could not map target file\n");
        return -1;
    }
    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);
    stub_size_aligned = align_to_boundary(stub_size, nt_headers-
>OptionalHeader.SectionAlignment);
    const char *section_name = ".inject";
    file_info_destroy(target_file);
    target_file = file_info_create();
    (void)map_file(target_file_name, stub_size_aligned, true, target_file);
    PIMAGE_SECTION_HEADER new_section = add_section(section_name,
stub_size_aligned, target_file->file_mem_buffer);
    if(new_section == NULL) {
        wprintf(L"Could not add new section to file");
        return -1;
    }
```

```
    write_stub_entry_point(nt_headers, stub_addr);
    copy_stub_instructions(new_section, target_file->file_mem_buffer,
stub_addr);
    change_file_oep(nt_headers, new_section);
    encrypt_file(nt_headers, target_file, section_name);
    int flush_view_success = FlushViewOfFile(target_file->file_mem_buffer,
0);
    if(flush_view_success == 0)
        wprintf(L"Could not save changes to file");
    file_info_destroy(target_file);
    return 0;
}
```

Injector.cpp

```
#include "Injector.h"
#include <stdio.h>

//Assumes malloc won't fail
pfile_info file_info_create(void) {
    pfile_info mapped_file_info = (pfile_info)malloc(sizeof(file_info));
    memset(mapped_file_info, 0, sizeof(file_info));
    return mapped_file_info;
}

//Assumes everything is valid, doesn't report error code
void file_info_destroy(pfile_info mapped_file_info) {
    if(mapped_file_info->file_mem_buffer != NULL)
        UnmapViewOfFile(mapped_file_info->file_mem_buffer);
    if(mapped_file_info->file_handle != NULL)
        CloseHandle(mapped_file_info->file_handle);
    if(mapped_file_info->file_map_handle != NULL)
        CloseHandle(mapped_file_info->file_map_handle);
    free(mapped_file_info);
    mapped_file_info = NULL;
}

inline unsigned int align_to_boundary(unsigned int address, unsigned int
boundary) {
        return (((address + boundary - 1) / boundary) * boundary);
}
unsigned int get_stub_size(void* stub_addr) {
    unsigned int size = 0;
    if(stub_addr != NULL) {
        const char *stub_signature = "\xCC\xCC\xCC\xCC";
        while(memcmp(((unsigned char *)stub_addr + size), stub_signature,
sizeof(int)) != 0)
            ++size;
    }
    return size;
}

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;
}

//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;
}

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);
}

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;
}

void write_stub_entry_point(PIMAGE_NT_HEADERS nt_headers, void *stub_addr) {
    if(stub_addr != NULL) {
        const char *signature = "\xFF\xEE\xDD\xCC";
        unsigned int index = 0;
        while(memcmp(((unsigned char *)stub_addr + index), signature,
sizeof(int)) != 0) {
            ++index;
        }
        DWORD old_protections = 0;
        VirtualProtect(((unsigned char *)stub_addr + index), sizeof(DWORD),
PAGE_EXECUTE_READWRITE, &old_protections);
        memcpy(((unsigned char *)stub_addr + index), &nt_headers-
>OptionalHeader.AddressOfEntryPoint, sizeof(DWORD));
        VirtualProtect(((unsigned char *)stub_addr + index), sizeof(DWORD),
old_protections, NULL);
```

```
    }
}
```

Injector.h

```
#pragma once
#include <Windows.h>

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

pfile_info file_info_create(void);
void file_info_destroy(pfile_info mapped_file_info);
unsigned int align_to_boundary(unsigned int address, unsigned int boundary);
unsigned int get_stub_size(void* stub_addr);
bool map_file(const wchar_t *file_name, unsigned int stub_size, bool
append_mode, pfile_info mapped_file_info);
PIMAGE_SECTION_HEADER add_section(const char *section_name, unsigned int
section_size, void *image_addr);
void copy_stub_instructions(PIMAGE_SECTION_HEADER section, void *image_addr,
void *stub_addr);
void change_file_oep(PIMAGE_NT_HEADERS nt_headers, PIMAGE_SECTION_HEADER
section);
void write_stub_entry_point(PIMAGE_NT_HEADERS nt_headers, void *stub_addr);
```

Encrypter.cpp

```
#include "Encrypter.h"
#include <stdio.h>

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]);
    }
}

//Unused, kept for testing/verification
void decrypt(unsigned int num_rounds, unsigned int blocks[2], unsigned int
const key[4]) {
    const unsigned int delta = 0x9E3779B9;
    unsigned int sum = delta * num_rounds;
    for (unsigned int i = 0; i < num_rounds; ++i) {
        blocks[1] -= (((blocks[0] << 4) ^ (blocks[0] >> 5)) + blocks[0]) ^
(sum + key[(sum >> 11) & 3]);
        sum -= delta;
        blocks[0] -= (((blocks[1] << 4) ^ (blocks[1] >> 5)) + blocks[1]) ^
(sum + key[sum & 3]);
    }
}

inline unsigned int swap_endianess(unsigned int value) {
    return (value >> 24) |  ((value << 8) & 0x00FF0000) |
        ((value >> 8) & 0x0000FF00) | (value << 24);
}
```

Encrypter.h

```
#pragma once
#include "Injector.h"

void encrypt_file(PIMAGE_NT_HEADERS nt_headers, pfile_info target_file, const
char *excluded_section_name);
```

```
void encrypt(unsigned int num_rounds, unsigned int blocks[2], unsigned int
const key[4]);
void decrypt(unsigned int num_rounds, unsigned int blocks[2], unsigned int
const key[4]);
unsigned int swap_endianess(unsigned int value);
```

A few general remarks about the code:

- Programs utilizing TLS callbacks may or may not work properly (depending on what the callbacks do). Full support for TLS callbacks can be implemented without issue
- An interesting idea would be to decrypt sections or pages as needed. This could be done by setting memory breakpoints on the sections or on individual pages. The instructions can be encrypted again afterwards once they've executed. This requires quite a bit of work in implementing a SEH handler in assembly and registering the exception in the processes exception list.
- This code only works on x86 executables. This is extremely obvious and not much can be done in that regard.
- The source needs to be built in release mode with any sort of extra optimizations/security (ESP checking/security cookies) disabled.