From a C project, through assembly, to shellcode

v 1.2

by hasherezade for @vxunderground

special thanks to Duchy for testing

Table of Contents

Introduction

Malware authors (as well as exploit developers) often use in their work pieces of a standalone, position independent code - referred to as shellcodes. This type of code can be injected very easily in any fitting place of the memory, and executed immediately - without a need for external loaders. Although shellcode offers many advantages for researchers (and malware authors), crafting it is tedious. Shellcodes must follow a very different set of principles than the formats that are the usual output of a compiler. That's why, usually people write them in assembly in order to have full control over the format of the generated output.

Creating shellcodes in assembly is the surest and the most accurate way. Yet, it is tedious and error-prone. That's why various researchers come up with their own ideas of simplifying the whole process, and taking advantage of the C compiler rather than crafting full shellcode by hand in assembly. In this document I will share my experience about it, and the method I use for those purposes.

This paper is intended to be beginner-friendly, so I described in detail some well-known, generic techniques around shellcode creation. In the first paragraphs, I show some general principles that shellcodes need to follow, and the reasoning behind the presented method. Then, I provide step-by-step walk-troughs and examples of shellcodes created with its help.

With the presented method, we can avoid writing the assembly full by ourselves - yet, we will be able to conveniently edit the generated assembly. We don't lose the advantages of handcrafting the shellcode, yet we skip the tedious part.

Prior work and motivations

The idea of creating shellcodes from C code is not new.

In "The Rootkit Arsenal - Second Edition" from 2012 Bill Blunden explains his way of creating shellcodes from C code (*Chapter 10: Building Shellcode in C*). A similar method was described by Matt Graeber (Mattifestation) in his "Writing Optimized Windows Shellcode in C" article. In both cases, the shellcode was created directly from a C code, and the whole idea was related to changing the compiler settings in order to create a PE file from which we are able to extract the buffer of independent code.

Still, what I missed in those methods, were the advantages of the shellcode written in pure assembly from scratch. In the above cases, we could only get the final code - but we had no direct control on the generated assembly, and no opportunity to interact with it or do changes.

What I was looking for was a method that will connect the best of both worlds: allowing to omit the tedious and error-prone part of writing the assembly. Yet, generating the assembly that I could freely tinker with, and finally use to generate my shellcode.

Shellcode - general principles

In case of PE format we just write a code and don't have to worry how it is loaded: Windows Loader will do it. It is different when we write shellcode. We cannot rely on the conveniences provided by PE format and Windows Loader:

- No sections
- No Data Directories (imports, relocations)

We have only the code to provide everything we need…

Overview of some important differences between the PE and the shellcode:

Feature	PE file	Shellcode
Loading	via Windows Loader; running new EXE triggers creation of a new process	Custom, simplified; must parasite on existing process (i.e. via code injection + thread injection), or appended to an existing PE (i.e. in case of a virus)
Composition	Sections with specific access rights, carrying various elements (code, data, resources, etc)	All in one memory area (read,write,execute)
Relocation to the load base	Defined by Relocation Table, applied by Windows Loader	Custom; position-independent code
Access to system API (Imports loading)	Defined by the Import Table, applied by Windows Loader	Custom: retrieving imports via PEB lookup; no IAT, or simplified

Position-independent code

In the case of PE files, we have a relocation table that is used by Windows Loader to shift all the addresses accordingly to the base where the executable was loaded in the memory. It is done automatically at runtime.

In case of shellcodes, we cannot take advantage of this feature - so we just need to write the code in the way that no relocation will be required. A code that follows those principles is called a Position Independent Code (PIC).

We create a position independent code by using only addresses that are relative to the current instruction pointer. We can use short jumps, long jumps, calls to local functions - because all of them are relative.

Let assume, as one of the steps of creating the shellcode, we will create a PE which's full code sections is a Position-Independent Code. To achieve this, we cannot use any addresses that reference data from other PE sections. If we want to use any strings, or other data, we must inline it in the code.

Calling API without the Import Table

In case of PE, all the API calls that we referenced in the code will be gathered in the Import Table. Creation of the Import Table is done by the linker. Then, resolving the Import Table is done at runtime. All is handled by default.

In case of shellcodes we no longer can access the Import Table, so we need to take care of resolving the APIs by ourselves.

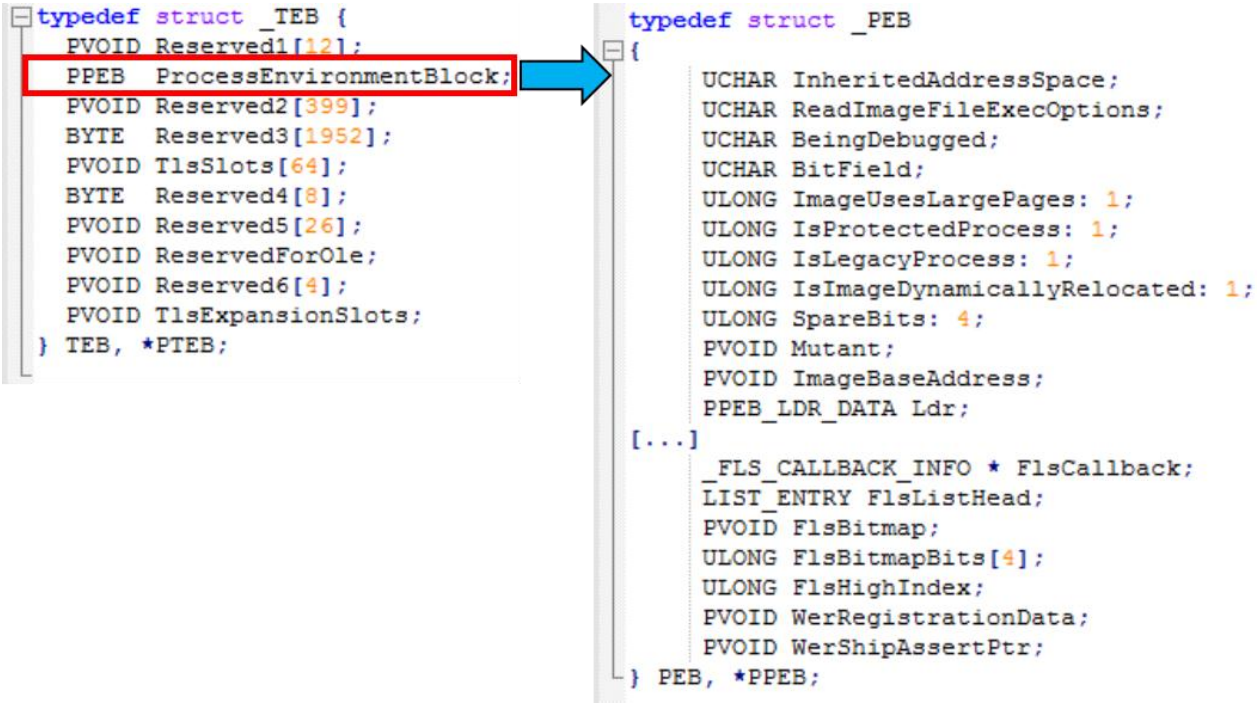
In order to retrieve the API functions that we use in our shellcode, we will take advantage of the PEB (Process Environment Block - one of the system structures that is created at process' runtime). Once our shellcode is injected into a process, we will retrieve the PEB of the target, and then use it to search for DLLs that are loaded in the process address space. We fetch Ntdll.dll or Kernel32.dll in order to resolve the rest of the imports. Ntdll.dll is loaded in every process at its early stage. Kernel32.dll is loaded in majority of the processes after the initialization – so we can safely assume that it will be loaded in the process of our interest. Once we retrieve any of them, we can use it to load other needed DLLs.

Overview of retrieving Imports for a shellcode:

1. Get the PEB address
2. Via PEB->`Ldr`->`InMemoryOrderModuleList`, find:
   - kernel32.dll (in majority of the processes it is loaded by default)
   - or ntdll.dll (if we want to use low-level equivalents of Import loading functions)
3. Walk through exports table of `kernel32` (or `ntdll`) to find addresses of:
   - `kernel32.LoadLibraryA` (eventually: `ntdll.LdrLoadDll`)
   - `kernel32.GetProcAddress` (eventually: `ntdll.LdrGetProcedureAddress`)
4. Use `LoadLibraryA` (or `LdrLoadDll`) to load a needed DLL
5. Use `GetProcAddress` (or `LdrGetProcedureAddress`) to retrieve a needed function

Retrieving PEB

Fortunately, PEB can be retrieved easily by pure assembly. A pointer to PEB is one of the fields of another structure: TEB (Thread Environment Block).



```
typedef struct _TEB {
    PVOID Reserved1[12];
    PPEB  ProcessEnvironmentBlock;
    PVOID Reserved2[399];
    BYTE  Reserved3[1952];
    PVOID TlsSlots[64];
    BYTE  Reserved4[8];
    PVOID Reserved5[26];
    PVOID ReservedForOle;
    PVOID Reserved6[4];
    PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

```
typedef struct _PEB
{
    UCHAR InheritedAddressSpace;
    UCHAR ReadImageFileExecOptions;
    UCHAR BeingDebugged;
    UCHAR BitField;
    ULONG ImageUsesLargePages: 1;
    ULONG IsProtectedProcess: 1;
    ULONG IsLegacyProcess: 1;
    ULONG IsImageDynamicallyRelocated: 1;
    ULONG SpareBits: 4;
    PVOID Mutant;
    PVOID ImageBaseAddress;
    PPEB_LDR_DATA Ldr;
[...]
    _FLS_CALLBACK_INFO * FlsCallback;
    LIST_ENTRY FlsListHead;
    PVOID FlsBitmap;
    ULONG FlsBitmapBits[4];
    ULONG FlsHighIndex;
    PVOID WerRegistrationData;
    PVOID WerShipAssertPtr;
} PEB, *PPEB;
```

The TEB is pointed by a segment register: FS in case of a 32 bit process, and GS in case of 64 bit process.

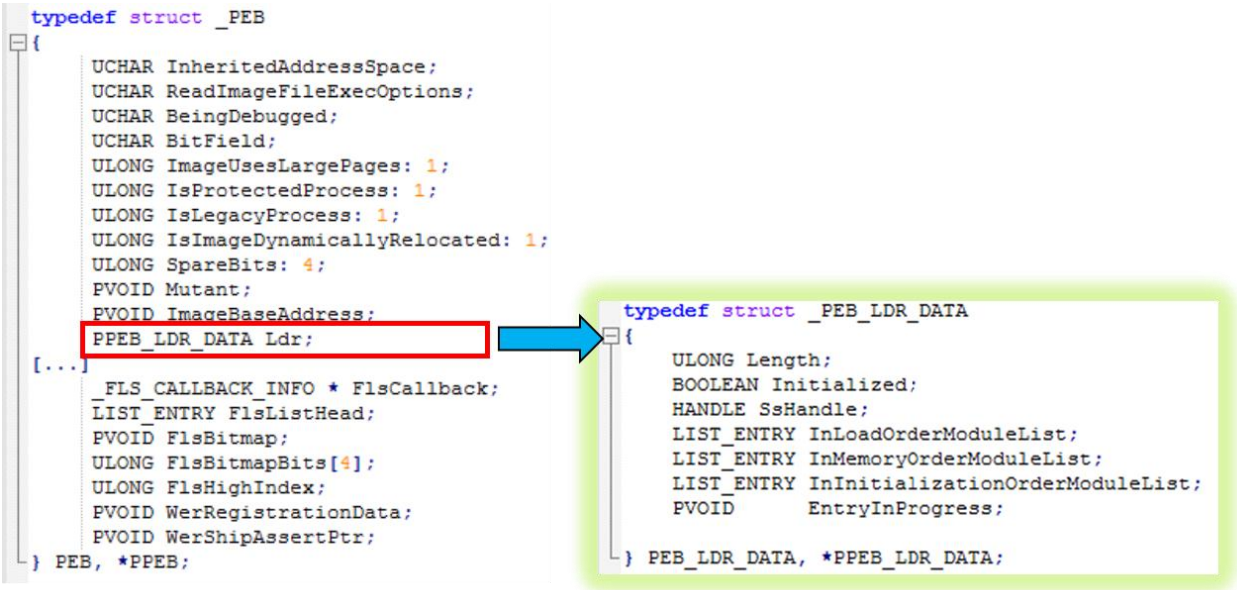
process bitness	**32 bit**	**64 bit**
pointer to TEB	FS register	GS register
offset to PEB from TEB	0x30	0x60

In order to get PEB from assembly, we just fetch a field at a particular offset relative to the segment register pointing to TEB. If we implement it in C, it looks in the following way:

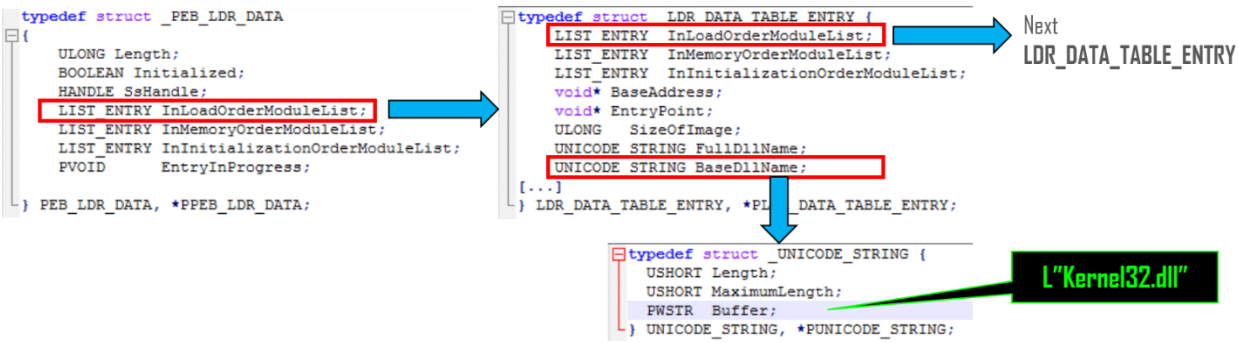
```
    PPEB peb = NULL;
#if defined(_WIN64)
    peb = (PPEB)__readgsqword(0x60);
#else
    peb = (PPEB)__readfsdword(0x30);
#endif
```

PEB-based DLL lookup

One of the PEB's fields is a linked list of all the DLLs that are loaded in the memory of the process:



We will walk through this list, till we find the DLL we are looking for.



At this point we need a DLL that will help us to resolve other APIs that we want to import. We can do it with the help of Kernel32.dll (or eventually Ntdll.dll, yet, Kernel32 is more handy).

The whole process of retrieving a DLL with chosen name by DLL lookup is demonstrated in the following C code:

```
#include <Windows.h>

#ifndef __NTDLL_H__

#ifndef TO_LOWERCASE
#define TO_LOWERCASE(out, c1) (out = (c1 <= 'Z' && c1 >= 'A') ? c1 = (c1 - 'A') + 'a': c1)
#endif


typedef struct _UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;

} UNICODE_STRING, * PUNICODE_STRING;

typedef struct _PEB_LDR_DATA
{
    ULONG Length;
    BOOLEAN Initialized;
    HANDLE SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID      EntryInProgress;

} PEB_LDR_DATA, * PPEB_LDR_DATA;

//here we don't want to use any functions imported form extenal modules

typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY  InLoadOrderModuleList;
    LIST_ENTRY  InMemoryOrderModuleList;
    LIST_ENTRY  InInitializationOrderModuleList;
    void* BaseAddress;
    void* EntryPoint;
    ULONG   SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG   Flags;
    SHORT   LoadCount;
    SHORT   TlsIndex;
    HANDLE  SectionHandle;
    ULONG   CheckSum;
    ULONG   TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, * PLDR_DATA_TABLE_ENTRY;

typedef struct _PEB
{
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    BOOLEAN SpareBool;
    HANDLE Mutant;

    PVOID ImageBaseAddress;
    PPEB_LDR_DATA Ldr;

    // [...] this is a fragment, more elements follow here
```

```
} PEB, * PPEB;

#endif //__NTDLL_H__

inline LPVOID get_module_by_name(WCHAR* module_name)
{
    PPEB peb = NULL;
#if defined(_WIN64)
    peb = (PPEB)__readgsqword(0x60);
#else
    peb = (PPEB)__readfsdword(0x30);
#endif
    PPEB_LDR_DATA ldr = peb->Ldr;
    LIST_ENTRY list = ldr->InLoadOrderModuleList;

    PLDR_DATA_TABLE_ENTRY Flink = *((PLDR_DATA_TABLE_ENTRY*)(&list));
    PLDR_DATA_TABLE_ENTRY curr_module = Flink;

    while (curr_module != NULL && curr_module->BaseAddress != NULL) {
        if (curr_module->BaseDllName.Buffer == NULL) continue;
        WCHAR* curr_name = curr_module->BaseDllName.Buffer;

        size_t i = 0;
        for (i = 0; module_name[i] != 0 && curr_name[i] != 0; i++) {
            WCHAR c1, c2;
            TO_LOWERCASE(c1, module_name[i]);
            TO_LOWERCASE(c2, curr_name[i]);
            if (c1 != c2) break;
        }
        if (module_name[i] == 0 && curr_name[i] == 0) {
            //found
            return curr_module->BaseAddress;
        }
        // not found, try next:
        curr_module = (PLDR_DATA_TABLE_ENTRY)curr_module->InLoadOrderModuleList.Flink;
    }
    return NULL;
}
```

Exports lookup

Once we retrieved the base of Kernel32.dll, we still need to retrieve addresses of the needed functions: `LoadLibraryA` and `GetProcAddress`. We will do it by exports lookup.

First we need to fetch the Exports Table from the Data Directory of the found DLL. Then we walk through all the functions exported by names, till we find the name of our interest. We fetch the RVA associated with the name, and add the module base, to get the absolute address (VA).

The exports lookup function:

```
inline LPVOID get_func_by_name(LPVOID module, char* func_name)
{
    IMAGE_DOS_HEADER* idh = (IMAGE_DOS_HEADER*)module;
    if (idh->e_magic != IMAGE_DOS_SIGNATURE) {
        return NULL;
    }
    IMAGE_NT_HEADERS* nt_headers = (IMAGE_NT_HEADERS*)((BYTE*)module + idh->e_lfanew);
    IMAGE_DATA_DIRECTORY* exportsDir = &(nt_headers-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
    if (exportsDir->VirtualAddress == NULL) {
        return NULL;
    }

    DWORD expAddr = exportsDir->VirtualAddress;
    IMAGE_EXPORT_DIRECTORY* exp = (IMAGE_EXPORT_DIRECTORY*)(expAddr + (ULONG_PTR)module);
    SIZE_T namesCount = exp->NumberOfNames;

    DWORD funcsListRVA = exp->AddressOfFunctions;
    DWORD funcNamesListRVA = exp->AddressOfNames;
    DWORD namesOrdsListRVA = exp->AddressOfNameOrdinals;

    //go through names:
    for (SIZE_T i = 0; i < namesCount; i++) {
        DWORD* nameRVA = (DWORD*)(funcNamesListRVA + (BYTE*)module + i * sizeof(DWORD));
        WORD* nameIndex = (WORD*)(namesOrdsListRVA + (BYTE*)module + i * sizeof(WORD));
        DWORD* funcRVA = (DWORD*)(funcsListRVA + (BYTE*)module + (*nameIndex) * sizeof(DWORD));

        LPSTR curr_name = (LPSTR)(*nameRVA + (BYTE*)module);
        size_t k = 0;
        for (k = 0; func_name[k] != 0 && curr_name[k] != 0; k++) {
            if (func_name[k] != curr_name[k]) break;
        }
        if (func_name[k] == 0 && curr_name[k] == 0) {
            //found
            return (BYTE*)module + (*funcRVA);
        }
    }
    return NULL;
}
```

Wrapping up: the header

We will gather all the above code in a header `peb_lookup.h` (available here), that we can include to our projects where we want to make use of the PEB lookup.

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

#ifndef __NTDLL_H__

#ifndef TO_LOWERCASE
#define TO_LOWERCASE(out, c1) (out = (c1 <= 'Z' && c1 >= 'A') ? c1 = (c1 - 'A') + 'a': c1)
#endif
```

```
typedef struct _UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;

} UNICODE_STRING, * PUNICODE_STRING;

typedef struct _PEB_LDR_DATA
{
    ULONG Length;
    BOOLEAN Initialized;
    HANDLE SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID      EntryInProgress;

} PEB_LDR_DATA, * PPEB_LDR_DATA;

//here we don't want to use any functions imported form external modules

typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY  InLoadOrderModuleList;
    LIST_ENTRY  InMemoryOrderModuleList;
    LIST_ENTRY  InInitializationOrderModuleList;
    void* BaseAddress;
    void* EntryPoint;
    ULONG   SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG   Flags;
    SHORT   LoadCount;
    SHORT   TlsIndex;
    HANDLE  SectionHandle;
    ULONG   CheckSum;
    ULONG   TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, * PLDR_DATA_TABLE_ENTRY;


typedef struct _PEB
{
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    BOOLEAN SpareBool;
    HANDLE Mutant;

    PVOID ImageBaseAddress;
    PPEB_LDR_DATA Ldr;

    // [...] this is a fragment, more elements follow here

} PEB, * PPEB;

#endif //__NTDLL_H__
```

```
inline LPVOID get_module_by_name(WCHAR* module_name)
{
    PPEB peb = NULL;
#if defined(_WIN64)
    peb = (PPEB)__readgsqword(0x60);
#else
    peb = (PPEB)__readfsdword(0x30);
#endif
    PPEB_LDR_DATA ldr = peb->Ldr;
    LIST_ENTRY list = ldr->InLoadOrderModuleList;

    PLDR_DATA_TABLE_ENTRY Flink = *((PLDR_DATA_TABLE_ENTRY*)(&list));
    PLDR_DATA_TABLE_ENTRY curr_module = Flink;

    while (curr_module != NULL && curr_module->BaseAddress != NULL) {
        if (curr_module->BaseDllName.Buffer == NULL) continue;
        WCHAR* curr_name = curr_module->BaseDllName.Buffer;

        size_t i = 0;
        for (i = 0; module_name[i] != 0 && curr_name[i] != 0; i++) {
            WCHAR c1, c2;
            TO_LOWERCASE(c1, module_name[i]);
            TO_LOWERCASE(c2, curr_name[i]);
            if (c1 != c2) break;
        }
        if (module_name[i] == 0 && curr_name[i] == 0) {
            //found
            return curr_module->BaseAddress;
        }
        // not found, try next:
        curr_module = (PLDR_DATA_TABLE_ENTRY)curr_module->InLoadOrderModuleList.Flink;
    }
    return NULL;
}



inline LPVOID get_func_by_name(LPVOID module, char* func_name)
{
    IMAGE_DOS_HEADER* idh = (IMAGE_DOS_HEADER*)module;
    if (idh->e_magic != IMAGE_DOS_SIGNATURE) {
        return NULL;
    }
    IMAGE_NT_HEADERS* nt_headers = (IMAGE_NT_HEADERS*)((BYTE*)module + idh->e_lfanew);
    IMAGE_DATA_DIRECTORY* exportsDir = &(nt_headers-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
    if (exportsDir->VirtualAddress == NULL) {
        return NULL;
    }

    DWORD expAddr = exportsDir->VirtualAddress;
    IMAGE_EXPORT_DIRECTORY* exp = (IMAGE_EXPORT_DIRECTORY*)(expAddr + (ULONG_PTR)module);
    SIZE_T namesCount = exp->NumberOfNames;

    DWORD funcsListRVA = exp->AddressOfFunctions;
    DWORD funcNamesListRVA = exp->AddressOfNames;
    DWORD namesOrdsListRVA = exp->AddressOfNameOrdinals
```

```
    //go through names:
    for (SIZE_T i = 0; i < namesCount; i++) {
        DWORD* nameRVA = (DWORD*)(funcNamesListRVA + (BYTE*)module + i * sizeof(DWORD));
        WORD* nameIndex = (WORD*)(namesOrdsListRVA + (BYTE*)module + i * sizeof(WORD));
        DWORD* funcRVA = (DWORD*)(funcsListRVA + (BYTE*)module + (*nameIndex) * sizeof(DWORD));

        LPSTR curr_name = (LPSTR)(*nameRVA + (BYTE*)module);
        size_t k = 0;
        for (k = 0; func_name[k] != 0 && curr_name[k] != 0; k++) {
            if (func_name[k] != curr_name[k]) break;
        }
        if (func_name[k] == 0 && curr_name[k] == 0) {
            //found
            return (BYTE*)module + (*funcRVA);
        }
    }
    return NULL;
}
```

Writing and compiling code in assembly

As mentioned before, the typical approach for writing shellcodes is to do it in assembly.

When we write assembly, we first have to choose the assembler that we want to compile the code with. This choice imposes some subtle differences on the syntax that we have to use.

The most popular assembler for Windows is **MASM** - which is a part of Visual Studio, and comes in two versions: 32 bit (ml.exe) and 64 bit (ml64.exe). The output generated by MASM is an object file that can be linked to a PE. Let assume we have a simple example written in 32 bit MASM that displays a message box:

```
.386
.model flat

extern _MessageBoxA@16:near
extern _ExitProcess@4:near

.data
msg_title db "Demo!", 0
msg_content db "Hello World!", 0

.code
main proc
        push    0
        push    0
        push    offset msg_title
        push    offset msg_content
        push    0
        call    _MessageBoxA@16
        push    0
        call    _ExitProcess@4
main endp
end
```

We will compile this code by:

```
ml /c demo32.asm
```

And then we can link it by a default Visual Studio linker:

```
link demo32.obj /subsystem:console /defaultlib:kernel32.lib
/defaultlib:user32.lib /entry:main /out:demo32_masm.exe
```

Sometimes we can also deploy compiling and linking in one go, by using it without any parameters:

```
ml  demo32.asm
```

MASM is the default assembler for Windows. However, the most popular choice to create shellcodes is another assembler: **YASM** (the successor of NASM). It is a free, independent assembler for multiple platforms. It can be used to create PE files just like MASM. The syntax of YASM is a bit different. Let assume we have an analogical example written in 32 bit YASM:

```
bits 32

extern _MessageBoxA@16:proc
extern _ExitProcess@4:proc

msg_title db "Demo!", 0
msg_content db "Hello World!", 0

global main

main:
        push    0
        push    0
        push    msg_title
        push    msg_content
        push    0
        call    _MessageBoxA@16
        push    0
        call    _ExitProcess@4
```

We can compile it by:

```
yasm -f win32 demo32.asm -o demo32.obj
```

And, similarly to the MASM code, link it with Visual Studio linker (or any other linker of our choice):

```
link demo32.obj /defaultlib:user32.lib /defaultlib:kernel32.lib
/subsystem:windows /entry:main /out:demo32_yasm.exe
```

In contrast to MASM, YASM can be also used to compile the code into a binary file, rather than to the object file. So, we can get a ready to use binary buffer with the shellcode. Example of compiling to a binary file:

```
yasm -f bin demo.asm
```

Note that none of the above examples can be compiled to a shellcode, because they have external dependencies - so they don't follow the shellcode principles. Yet we could refactor them into a shellcode by removing the dependencies.

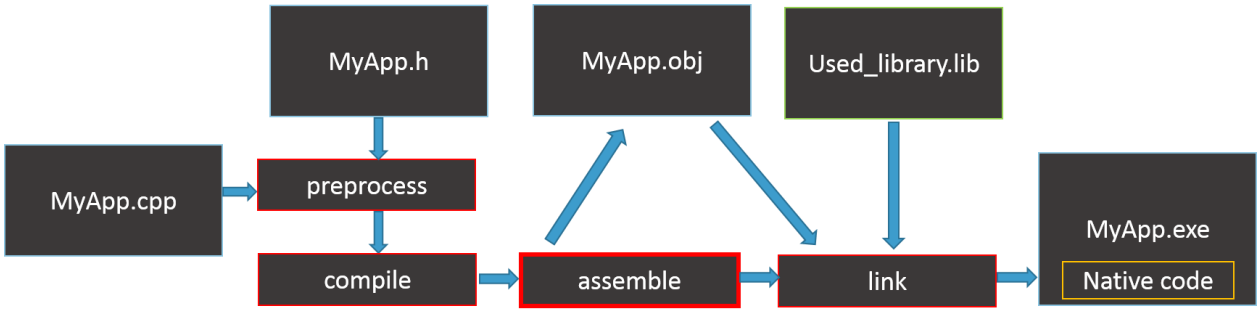
The method that is going to be presented in this paper uses MASM. The reason behind this choice is simple: if we generate the assembly code from the C file with the help of Visual Studio C compiler, the result will be in the MASM syntax. In contrast to YASM, cannot set it to output the shellcode directly: we will need to manually cut out the code from the PE. As we will see, although it may seem like a minor inconvenience, it has its pros, such as simplifies testing.

Compiling a C project - step by step

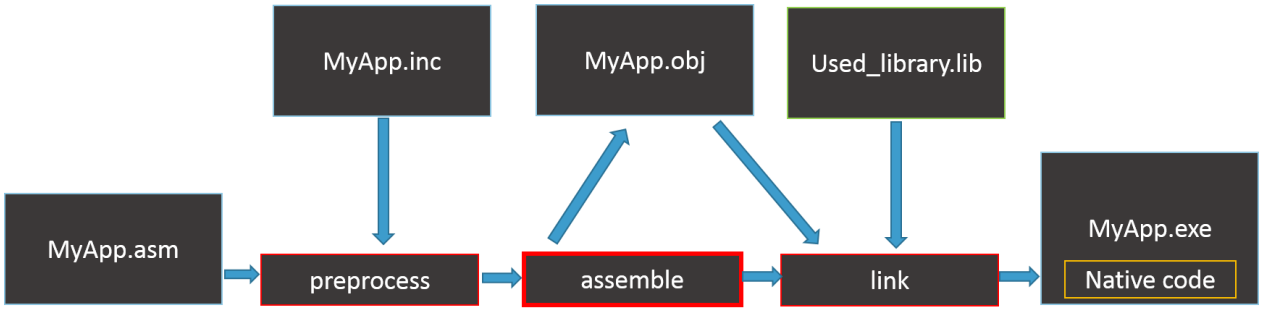
Nowadays, most of the people compile their code using integrated environments such as Visual Studio, which hides under the hood the whole compilation process. We just write a code, then compile it and link in one go. By default, the end result is a PE file: a native executable format for Windows.

Yet, sometimes it is useful to split this process into steps, so that we can have more control over it.

Lets' revise how the compilation of the C/C++ code looks at conceptual level:



And now compare it with the steps taken when the application is created from a code written in assembler:



As we can see, the compilation of the code from the higher level language differs just at the beginning. Also, while compiling the C code, in one of the steps assembly code is generated. This is interesting, because instead of writing the assembly by ourselves, we can write the

code in C, and then request the compiler to give us assembly as the output. Then, we will just need to modify the assembly according to the shellcode principles. More about it will be explained in further paragraphs.

We have the following example code:

```
#include <Windows.h>

int main()
{
    const char msg_title[] = "Demo!";
    const char msg_content[] = "Hello World!";

    MessageBoxA(0, msg_title, msg_content, MB_OK);
    ExitProcess(0);
}
```

Let's try to use Visual Studio compiler and linker from the command prompt, instead of the integrated environment. We can do it by selecting "VS Native Tools Command Prompt". Then we need to traverse to the directory with our code.

The bitness of the output executable (32 or 64 bit) will be selected by default depending what version of the command prompt you selected.

To compile the code, we use cl.exe. Using the option `/c` compiles code but prevents from linking it: so, as the result we get an object file (*.obj).

```
cl /c demo.cpp
```

Then, we may link the obj file with the help of the default linker that is a part of the Visual Studio package: link.exe. Sometimes we will need to provide additional libraries with which the application must be linked, or the entry point (in case if it uses a label different from the default one). Example of linking:

```
link demo.obj /defaultlib:user32.lib /out:demo_cpp.exe
```

Note that since the steps are independent from each other, you can also use an alternative linker instead of the default one - which also may be used to manipulate or obfuscate the format. A good example is crinkler, which is an executable compressor in the form of a linker. But this is another story…

If you add a parameter `/FA`, in addition to the `*.obj` file you will also get an assembly output in MASM.

```
cl /c /FA demo.cpp
```

You can then compile the generated assembly into an object file using MASM:

```
ml /c demo.asm
```

Dividing this process into steps gives us the opportunity to manipulate the assembly, and adjust it to our needs rather than writing it from scratch.

From a C project to the shellcode

The core idea

The presented method of creating shellcodes takes advantage of the fact that we can compile the C code into assembly. It consists of the following basic steps:

1. Prepare a project in C.
2. Refactor the project to load all the used imports by the PEB lookup (remove the dependency from the Import Table)
3. Use a C compiler to generate assembly:

```
cl /c /FA /GS- <file_name>.cpp
```

4. Refactor the assembly code to make it a valid shellcode (remove other left dependencies, inline the strings, variables, etc)
5. Compile it by MASM:

```
ml /c file.asm
```

6. Link it into a valid PE file, test if it runs properly
7. Dump the code section (i.e. with the help of PE-bear) - this is our shellcode

Note that the assembly generated by the C compiler is not guaranteed to be always 100% valid MASM code, because it is mostly generated as a listing for informational purposes. So, sometimes it will require manual cleaning.

Preparing the C project

When we prepare a C project to be compiled as a shellcode, we need to follow some rules: + do not use imports directly - always resolve them dynamically via PEB + do not use any static libraries + use only local variables: no global, no static (otherwise they will be stored in a separate section and break the Position-Independence!) + use stack-based strings (or inline them later in assembly)

As an example to illustrate the ides, we will use the simple demo, popping up a messagebox:

```
#include <Windows.h>

int main()
{
    MessageBoxW(0, L"Hello World!", L"Demo!", MB_OK);
    ExitProcess(0);
}
```

Preparing the Imports

As the first step of our preparation, we need to make all the used imports load dynamically. We have 2 imports in this project: `MessageBoxA` from `user32.dll` and `ExitProcess` from `kernel32.dll`.

In a normal case, if we want those imports to load dynamically, and not be included in the Import Table, we will refactor it similarly to the following:

```
#include <Windows.h>

int main()
{
    LPVOID u32_dll = LoadLibraryA("user32.dll");

    int (WINAPI * _MessageBoxW)(
        _In_opt_ HWND hWnd,
        _In_opt_ LPCWSTR lpText,
        _In_opt_ LPCWSTR lpCaption,
        _In_ UINT uType) = (int (WINAPI*)(
            _In_opt_ HWND,
            _In_opt_ LPCWSTR,
            _In_opt_ LPCWSTR,
            _In_ UINT)) GetProcAddress((HMODULE)u32_dll, "MessageBoxW");

    if (_MessageBoxW == NULL) return 4;

    _MessageBoxW(0, L"Hello World!", L"Demo!", MB_OK);

    return 0;
}
```

This is good as the first step of preparation, but not enough: still, we have two dependencies left: `LoadLibraryA` and `GetProcAddress`. Those two functions we need to resolve by the PEB lookup - we will use the `peb_lookup.h` that we created in the previous part. This is how the final result of the refactoring will look like popup.cpp:

```
#include <Windows.h>
#include "peb_lookup.h"

int main()
{
    LPVOID base = get_module_by_name((const LPWSTR)L"kernel32.dll");
    if (!base) {
        return 1;
    }

    LPVOID load_lib = get_func_by_name((HMODULE)base, (LPSTR)"LoadLibraryA");
    if (!load_lib) {
        return 2;
    }
    LPVOID get_proc = get_func_by_name((HMODULE)base, (LPSTR)"GetProcAddress");
    if (!get_proc) {
        return 3;
    }
    HMODULE(WINAPI * _LoadLibraryA)(LPCSTR lpLibFileName) = (HMODULE(WINAPI*)(LPCSTR))load_lib;
    FARPROC(WINAPI * _GetProcAddress)(HMODULE hModule, LPCSTR lpProcName)
        = (FARPROC(WINAPI*)(HMODULE, LPCSTR)) get_proc;
```

```
    LPVOID u32_dll = _LoadLibraryA("user32.dll");

    int (WINAPI * _MessageBoxW)(
        _In_opt_ HWND hWnd,
        _In_opt_ LPCWSTR lpText,
        _In_opt_ LPCWSTR lpCaption,
        _In_ UINT uType) = (int (WINAPI*)(
            _In_opt_ HWND,
            _In_opt_ LPCWSTR,
            _In_opt_ LPCWSTR,
            _In_ UINT)) _GetProcAddress((HMODULE)u32_dll, "MessageBoxW");

    if (_MessageBoxW == NULL) return 4;

    _MessageBoxW(0, L"Hello World!", L"Demo!", MB_OK);

    return 0;
}
```

Beware of the jump tables

If we use switch conditions in the code, they may get compiled as a jump table. This is an automatic optimization performed by the compiler. In a normal executable, it is a beneficial solution. But when we write a shellcode, we must beware of it, because it breaks position independence of the code: jump table is a structure that requires to be relocated.

Example of how the jump table looks in assembly:

```
$LN14@switch_sta:
    DD  $LN8@switch_sta
    DD  $LN6@switch_sta
    DD  $LN10@switch_sta
    DD  $LN4@switch_sta
    DD  $LN2@switch_sta
$LN13@switch_sta:
    DB  0
    DB  1
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  4
    DB  2
    DB  4
    DB  4
    DB  4
    DB  4
    DB  3
```

The decision if the jump table will be generated or not for a switch statement is taken by a compiler. For a small number of cases (less than 4) it is usually not generated. But if we want to check a bigger number of conditions, we must refactor the code to avoid a long

switch statement: either break checks into multiple functions, or replace them by if-else statements.

Example:

- this long switch statement will cause a generation of the jump table:

```
bool switch_state(char *buf, char *resp)
{
    switch (resp[0]) {
        case 0:
            if (buf[0] != '9') break;
            resp[0] = 'Y';
            return true;
        case 'Y':
            if (buf[0] != '3') break;
            resp[0] = 'E';
            return true;
        case 'E':
            if (buf[0] != '5') break;
            resp[0] = 'S';
            return true;
        case 'S':
            if (buf[0] != '8') break;
            resp[0] = 'D';
            return true;
        case 'D':
            if (buf[0] != '4') break;
            resp[0] = 'O';
            return true;
        case 'O':
            if (buf[0] != '7') break;
            resp[0] = 'N';
            return true;
        case 'N':
            if (buf[0] != '!') break;
            resp[0] = 'E';
            return true;
    }
    return false;
}
```

- however, if we break it into several segments, we can avoid the generation of the jump table:

```
bool switch_state(char *buf, char *resp)
{
    {
        switch (resp[0]) {
        case 0:
            if (buf[0] != '9') break;
            resp[0] = 'Y';
            return true;
        case 'Y':
            if (buf[0] != '3') break;
            resp[0] = 'E';
            return true;
        case 'E':
            if (buf[0] != '5') break;
            resp[0] = 'S';
```

```
            return true;
        }
    }
    {
        switch (resp[0]) {
        case 'S':
            if (buf[0] != '8') break;
            resp[0] = 'D';
            return true;
        case 'D':
            if (buf[0] != '4') break;
            resp[0] = 'O';
            return true;
        case 'O':
            if (buf[0] != '7') break;
            resp[0] = 'N';
            return true;
        }
    }
    {
        switch (resp[0]) {
        case 'N':
            if (buf[0] != '!') break;
            resp[0] = 'E';
            return true;
        }
    }
    return false;
}
```

- alternatively, we can just rewrite it using if-else:

```
bool switch_state(char *buf, char *resp)
{

    if (resp[0] == 0 && buf[0] == '9') {
        resp[0] = 'Y';
    }
    else if (resp[0] == 'Y' && buf[0] == '3') {
        resp[0] = 'E';
    }
    else if (resp[0] == 'E' && buf[0] == '5') {
        resp[0] = 'S';
    }
    else if (resp[0] == 'S' && buf[0] == '8') {
        resp[0] = 'D';
    }
    else if (resp[0] == 'D' && buf[0] == '4') {
        resp[0] = 'O';
    }
    else if (resp[0] == 'O' && buf[0] == '7') {
        resp[0] = 'N';
    }
    else if (resp[0] == 'N' && buf[0] == '!') {
        resp[0] = 'E';
    }
    return false;
}
```

Removing the implicit dependencies

We must also be careful not to introduce some implicit dependencies in our project. For example if we initialize a variable in the following way:

```
struct sockaddr_in sock_config = { 0 };
```

It will cause an implicit call to `memset` from an external library. In assembly we will see the dependency perpended with the keyword `EXTRN`:

```
EXTRN   _memset:PROC
```

In order to remove such dependencies, we need to initialize the structures in a different way. Either by our own functions, or by functions that are guaranteed to be inlined, such as `SecureZeroMemory` (mentioned here):

```
struct sockaddr_in sock_config;
SecureZeroMemory(&sock_config, sizeof(sock_config));
```

Preparing the strings (optional)

At this point we may also refactor all the used string into stack-based strings, as it was described by Nick Harbour in the following article. Example:

```
    char load_lib_name[] = {'L','o','a','d','L','i','b','r','a','r','y','A',0};
    LPVOID load_lib = get_func_by_name((HMODULE)base, (LPSTR)load_lib_name);
```

After compilation to assembly, the string will look in the following way:

```
; Line 10
    mov BYTE PTR _load_lib_name$[ebp], 76   ; 0000004cH
    mov BYTE PTR _load_lib_name$[ebp+1], 111    ; 0000006fH
    mov BYTE PTR _load_lib_name$[ebp+2], 97 ; 00000061H
    mov BYTE PTR _load_lib_name$[ebp+3], 100    ; 00000064H
    mov BYTE PTR _load_lib_name$[ebp+4], 76 ; 0000004cH
    mov BYTE PTR _load_lib_name$[ebp+5], 105    ; 00000069H
    mov BYTE PTR _load_lib_name$[ebp+6], 98 ; 00000062H
    mov BYTE PTR _load_lib_name$[ebp+7], 114    ; 00000072H
    mov BYTE PTR _load_lib_name$[ebp+8], 97 ; 00000061H
    mov BYTE PTR _load_lib_name$[ebp+9], 114    ; 00000072H
    mov BYTE PTR _load_lib_name$[ebp+10], 121   ; 00000079H
    mov BYTE PTR _load_lib_name$[ebp+11], 65    ; 00000041H
    mov BYTE PTR _load_lib_name$[ebp+12], 0
; Line 11
    lea eax, DWORD PTR _load_lib_name$[ebp]
```

This step is an alternative to inlining strings in assembler, which will be presented later. We can choose any method that we find more convenient. If we choose to use stack-based strings, this is how our code will look like after the refactoring:

```
#include <Windows.h>
#include "peb_lookup.h"

int main()
{
    wchar_t kernel32_dll_name[] = {'k','e','r','n','e','l','3','2','.','d','l','l', 0};
    LPVOID base = get_module_by_name((const LPWSTR)kernel32_dll_name);
```

```
    if (!base) {
        return 1;
    }

    char load_lib_name[] = {'L','o','a','d','L','i','b','r','a','r','y','A',0};
    LPVOID load_lib = get_func_by_name((HMODULE)base, (LPSTR)load_lib_name);
    if (!load_lib) {
        return 2;
    }
    char get_proc_name[] = {'G','e','t','P','r','o','c','A','d','d','r','e','s','s',0};
    LPVOID get_proc = get_func_by_name((HMODULE)base, (LPSTR)get_proc_name);
    if (!get_proc) {
        return 3;
    }
    HMODULE(WINAPI * _LoadLibraryA)(LPCSTR lpLibFileName) = (HMODULE(WINAPI*)(LPCSTR))load_lib;
    FARPROC(WINAPI * _GetProcAddress)(HMODULE hModule, LPCSTR lpProcName)
        = (FARPROC(WINAPI*)(HMODULE, LPCSTR)) get_proc;

    char user32_dll_name[] = {'u','s','e','r','3','2','.','d','l','l', 0};
    LPVOID u32_dll = _LoadLibraryA(user32_dll_name);

    char message_box_name[] = {'M','e','s','s','a','g','e','B','o','x','W', 0};
    int (WINAPI * _MessageBoxW)(
        _In_opt_ HWND hWnd,
        _In_opt_ LPCWSTR lpText,
        _In_opt_ LPCWSTR lpCaption,
        _In_ UINT uType) = (int (WINAPI*)(
            _In_opt_ HWND,
            _In_opt_ LPCWSTR,
            _In_opt_ LPCWSTR,
            _In_ UINT)) _GetProcAddress((HMODULE)u32_dll, message_box_name);

    if (_MessageBoxW == NULL) return 4;

    wchar_t msg_content[] = {'H','e','l','l','o', ' ', 'W','o','r','l','d','!', 0};
    wchar_t msg_title[] = {'D','e','m','o','!', 0};
    _MessageBoxW(0, msg_title, msg_content, MB_OK);

    return 0;
}
```

Using stack-based strings has its pros and cons. The advantage is that we can implement it from a C code, and we don't have to alter them in the assembly later. However, inlining strings in assembly can be automated (i.e. by this small utility), so it is not a big inconvenience (and it also makes obfuscating strings easier).

In this article I decided to present the second way: so, we do not change the strings in the C file, but instead post-process the assembly. Yet, the method using stack-based strings is presented as a reference. (Of course we may also use a mix of both methods: refactor some of the strings to stack-based, and inline the remaining ones).

Compiling to assembly

Now we are ready to compile this project into assembly. This step is the same for 32 and 64 bit version - the only difference is that we need to choose a different Visual Studio Native Tools Command Prompt (appropriately x86 or x64):

```
cl /c /FA /GS- demo.cpp
```

Remember to store the `peb_lookup.h` header in the same directory as the `demo.cpp` - and it will be included automatically.

The flag `/FA` is very important because it is responsible for generating the assembly listing that we will be further processing.

Disabling the cookie checks

The flag `/GS-` is responsible for disabling stack cookie checks. If we forget to use it, our code will contain the following external dependencies:

```
EXTRN   __GSHandlerCheck:PROC
EXTRN   __security_check_cookie:PROC
EXTRN   __security_cookie:QWORD
```

And references to them, for example:

```
    sub rsp, 664                ; 00000298H
    mov rax, QWORD PTR __security_cookie
    xor rax, rsp
```

...

```
    mov rcx, QWORD PTR __$ArrayPad$[rsp]
    xor rcx, rsp
    call    __security_check_cookie
    add rsp, 664                ; 00000298H
    pop rdi
    pop rsi
    ret 0
```

We can still remove them manually as demonstrated below - but it is recommended to just disable them at the compilation stage.

Change security cookie to `0`:

```
     sub rsp, 664                ; 00000298H
    mov rax, 0; QWORD PTR __security_cookie
    xor rax, rsp
```

And remove the line where the security cookie was checked:

```
     mov rcx, QWORD PTR __$ArrayPad$[rsp]
    xor rcx, rsp
    ;call   __security_check_cookie
    add rsp, 664                ; 00000298H
    pop rdi
    pop rsi
    ret 0
```

Refactoring the assembly

The described method can be used for creating 32 bit shellcodes as well as 64 bit. However, there are some subtle differences between those two, and the steps may differ. That's why they will be described separately.

> Most of the steps described here can be automated with the help of masm_shc utility. Yet, I recommend going through the full process manually at least once, in order to get better understanding.

32 bit

To start, we need to have a 32 bit assembly, generated by the `cl /c /FA /GS- demo.cpp` command run from the 32 bit version of the Visual Studio Native Tools Command Prompt.

0. Cleaning up the assembly

First let's use it as is, and test if you can get the output EXE. We will try to compile the assembly using 32 bit MASM:

```
ml <file_name>.asm
```

Since we use FS register, the assembler will print an error:

```
Error A2108: use of register assumed to ERROR
```

In order to silence it out, we need to add the following line on the top of our file (just after the assembly header):

```
assume fs:nothing
```

After this modification, the file should compile without issues.

Run the output and make sure that everything works fine. At this point we should get a valid EXE. Yet, it we load it into a PE viewer, (i.e. PE-bear), we will see that although we removed all the dependencies in our C code, there are still some in the resulting output. It still has an Import Table. It is because of some standard libraries that are linked by default. We need to get rid of them.

1. Removing the rest of the external dependencies

In this step we need to get rid of the remaining imports, which came from the automatically included static libraries.

Comment-out the following includes:

```
INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES
```

You can also comment-out the line including listing:

```
include listing.inc
```

In the previous step, the object file was linked with a static library LibCMT that contained the default entry point: `_mainCRTStartup`. Now we removed this dependency. So, our entry point will not be found by the linker. We need to link it giving the entry point explicitly:

```
ml /c <file_name>.asm
link <file_name>.obj /entry:main
```

or, in one line (deploying the default linker just after the compilation):

```
ml /c <file_name>.asm /link /entry:main
```

Check if everything works fine. Open the resulting PE in PE-bear. You will see that now the PE does not have an Import Table at all. Also the code is much smaller. The Entry Point starts exactly in our main function.

2. Making the code Position-Independent: processing strings

Note that this step can be omitted if all strings are refactored to stack-based, as described here.

To make the shellcode position-independent, we cannot have any data stored in a separate section. We can only use the `.text` section for everything. So far, our strings are stored in the `.data` section. So, we need to refactor the assembly code to inline them.

Example of inlining a string:

- we copy the string from the data segment, and paste just before the line where it was pushed on the stack. We push it on the stack by making a call after the string:

```
    call after_kernel32_str
DB  'k', 00H, 'e', 00H, 'r', 00H, 'n', 00H, 'e', 00H, 'l', 00H
    DB  '3', 00H, '2', 00H, '.', 00H, 'd', 00H, 'l', 00H, 'l', 00H, 00H
    DB  00H
    ORG $+2
after_kernel32_str:
    ;push   OFFSET $SG89718
```

  If our projects has many strings, it can be laborious to inline all of them by hand, so it can be done automatically with masm_shc.

After inlining all the strings we should compile the application again, by:

```
ml /c <file_name>.asm /link /entry:main
```

Sometimes inlining the strings will make a distance between instructions too big, and prevent short jumps. We can fix it easily by changing short jumps to long. Example:

- Before:

```
jmp SHORT $LN1@main
```

- After:

```
jmp $LN1@main
```

Alternatively, we may copy the instructions where the jump was leading into.

Example - instead of jumping to the end of function to terminate the branch, we can make an alternative ending:

```
 ;jmp    SHORT $LN1@main
; Line 183
mov esp, ebp
pop ebp
ret 0
```

Test the resulting executable. If it doesn't run, it means you committed some error while inlining strings.

> Remember, that now all the strings are in the `.text` section. So, if you are processing (i.e. editing, decoding) a string that is inlined, first you must set the `.text` section as writable (by changing the flag in the sections headers) - otherwise the EXE will crash. Once the shellcode is extracted from the EXE, it will be anyways loaded into RWX (readable, writeable, executable) memory - so from the shellcode point of view it makes no difference. More about it will be described in the further example.

3. Cuting-out and testing the shellcode.

- Open the final version of the app in PE-bear. Notice that now the exe should have no import table, as well as no relocations table.
- Dump the `.text` section from the file using PE-bear
- Test the shellcode, by running it by `runshc32.exe` from the masm_shc package
- If everything went fine, the shellcode should run the same way as the EXE

64 bit

To start, we need to have a 64 bit assembly, generated by the `cl /c /FA /GS- demo.cpp` command run from the 64 bit version of the Visual Studio Native Tools Command Prompt.

The stack alignment stub

In case of the 64 bit code, we may also need to ensure 16-byte stack alignment. This alignment is required if we want to use XMM instructions in our code. If we fail to align the stack as expected, our application will crash as soon as we attempt to use an XMM register. More details about it was described in @mattifestation's article, under the paragraph "Ensuring Proper Stack Alignment in 64-bit Shellcode".

The code proposed by @mattifestation to ensure this alignment:

```
_TEXT SEGMENT

; AlignRSP is a simple call stub that ensures that the stack is 16-byte aligned prior
; to calling the entry point of the payload. This is necessary because 64-bit functions
; in Windows assume that they were called with 16-byte stack alignment. When amd64
; shellcode is executed, you can't be assured that you stack is 16-byte aligned. For example,
; if your shellcode lands with 8-byte stack alignment, any call to a Win32 function will likely
; crash upon calling any ASM instruction that utilizes XMM registers (which require 16-byte)
; alignment.

AlignRSP PROC
 push rsi    ; Preserve RSI since we're stomping on it
 mov  rsi, rsp  ; Save the value of RSP so it can be restored
 and  rsp, 0FFFFFFFFFFFFFFF0h ; Align RSP to 16 bytes
 sub  rsp, 020h  ; Allocate homing space for ExecutePayload
 call main ; Call the entry point of the payload
 mov  rsp, rsi  ; Restore the original value of RSP
 pop  rsi    ; Restore RSI
 ret      ; Return to caller
AlignRSP ENDP

_TEXT ENDS
```

This code is a stub from which we are supposed to run our main function, in order to align the stack before any of our code is executed.

We should append it before the first `_TEXT SEGMENT` of our file. Once we add this stub, it should become our new entry point of the application:

```
ml64 <file.asm> /link /entry:AlignRSP
```

0. Cleaning up the assembly

First let's use it as is, and test if it can give us valid output. We will try to compile the assembly using 64 bit MASM (from 64 bit version of the Visual Studio Native Tools Command Prompt):

```
ml64 <file_name>.asm
```

This time we get several errors. It is due to the fact that the generated listing is not fully compatible with MASM, and we need to fix all the compatibility issues manually. We will get the similar list of errors:

```
shellcode_task_step1.asm(75) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(86) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(98) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(116) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(120) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(132) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(133) : error A2006:undefined symbol : FLAT
shellcode_task_step1.asm(375) : error A2027:operand must be a memory expression
shellcode_task_step1.asm(30) : error A2006:undefined symbol : $LN16
shellcode_task_step1.asm(31) : error A2006:undefined symbol : $LN16
shellcode_task_step1.asm(36) : error A2006:undefined symbol : $LN13
```

```
shellcode_task_step1.asm(37) : error A2006:undefined symbol : $LN13
shellcode_task_step1.asm(41) : error A2006:undefined symbol : $LN7
shellcode_task_step1.asm(42) : error A2006:undefined symbol : $LN7
```

- We need to manually remove the word `FLAT` from the asm file. Just replace `FLAT:` with nothing.
- We need to remove the `pdata` and `xdata` segments
- We need to fix the reference to `gs` register to `gs:[96]`

from:

```
mov rax, QWORD PTR gs:96
```

to:

```
mov rax, QWORD PTR gs:[96]
```

Now the file should assemble properly. Run the resulting executable and check it in PE-bear.

1. Removing the rest of the external dependencies

In this step we need to get rid of the remaining imports, which came from the automatically included static libraries.

Just like in 32-bit version, we need to comment-out the automatically-added includes:

```
INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES
```

If some functions have been added automatically from those libraries, we need to get rid of them as described in the analogical part about the 32 bit version.

Compile the file, giving the Entry Point explicitly:

```
ml64 /c <file_name>.asm /link /entry:<entry_function>
```

2. Making the code Position-Independent: processing strings

Note that this step can be omitted if all strings are refactored to stack-based, as described here.

Analogically to the 32-bit version, we need to remove all the references to sections other than `.text`. In this case it means inlining all the strings. It will be similar like in the 32-bit version, but this time arguments to the functions are supplied via registers, rather than being pushed on the stack. That's why you need to pop their offsets into appropriate registers.

Example of inlined string for the 64 bit version:

```
call after_msgbox_str
    DB  'MessageBoxW', 00H
after_msgbox_str:
    pop rdx
    ;lea    rdx, OFFSET $SG90389
    mov rcx, QWORD PTR u32_dll$[rsp]
    call    QWORD PTR _GetProcAddress$[rsp]
```

3. Cutting-out and testing the shellcode - analogical to the 32-bit version:

- Open the final version of the app in PE-bear. Notice that now the exe should have no import table, as well as no relocations table.
- Dump the `.text` section from the file using PE-bear
- Test the shellcode, by running it by `runshc64.exe` from the masm_shc package
- If everything went fine, the shellcode should run the same way as the EXE

Extended example - a demo server

So far we prepared a little demo example, that was showing a MessageBox. But what about something more functional? Will it also work?

In this chapter we will have a look at another example - a little local server. It is a part of code from the White Rabbit crackme. This part opens sockets at 3 consecutive ports - one by one - and we are supposed to knock to those ports.

This is a C code knock.cpp that we can compile to assembly:

```
#include <Windows.h>
#include "peb_lookup.h"

#define LOCALHOST_ROT13 ">?D;=;=;>"

typedef struct
{
    HMODULE(WINAPI * _LoadLibraryA)(LPCSTR lpLibFileName);
    FARPROC(WINAPI * _GetProcAddress)(HMODULE hModule, LPCSTR lpProcName);
} t_mini_iat;

typedef struct
{
    int (PASCAL FAR *_WSAStartup)(
        _In_ WORD wVersionRequired,
        _Out_ LPWSADATA lpWSAData);

    SOCKET(PASCAL FAR *_socket)(
        _In_ int af,
        _In_ int type,
        _In_ int protocol);

    unsigned long (PASCAL FAR *_inet_addr)(_In_z_ const char FAR * cp);

    int (PASCAL FAR *_bind)(
        _In_ SOCKET s,
```

```
        _In_reads_bytes_(namelen) const struct sockaddr FAR *addr,
        _In_ int namelen);

    int (PASCAL FAR *_listen)(
        _In_ SOCKET s,
        _In_ int backlog);

    SOCKET(PASCAL FAR *_accept)(
        _In_ SOCKET s,
        _Out_writes_bytes_opt_(*addrlen) struct sockaddr FAR *addr,
        _Inout_opt_ int FAR *addrlen);

    int (PASCAL FAR *_recv)(
        _In_ SOCKET s,
        _Out_writes_bytes_to_(len, return) __out_data_source(NETWORK) char FAR * buf,
        _In_ int len,
        _In_ int flags);

    int (PASCAL FAR *_send)(
        _In_ SOCKET s,
        _In_reads_bytes_(len) const char FAR * buf,
        _In_ int len,
        _In_ int flags);

    int (PASCAL FAR *_closesocket)(IN SOCKET s);

    u_short(PASCAL FAR *_htons)(_In_ u_short hostshort);

    int (PASCAL FAR *_WSACleanup)(void);

} t_socket_iat;


bool init_iat(t_mini_iat &iat)
{
    LPVOID base = get_module_by_name((const LPWSTR)L"kernel32.dll");
    if (!base) {
        return false;
    }

    LPVOID load_lib = get_func_by_name((HMODULE)base, (LPSTR)"LoadLibraryA");
    if (!load_lib) {
        return false;
    }
    LPVOID get_proc = get_func_by_name((HMODULE)base, (LPSTR)"GetProcAddress");
    if (!get_proc) {
        return false;
    }

    iat._LoadLibraryA = (HMODULE(WINAPI*)(LPCSTR)) load_lib;
    iat._GetProcAddress = (FARPROC(WINAPI*)(HMODULE, LPCSTR)) get_proc;
    return true;
}

bool init_socket_iat(t_mini_iat &iat, t_socket_iat &sIAT)
{
    LPVOID WS232_dll = iat._LoadLibraryA("WS2_32.dll");

    sIAT._WSAStartup = (int (PASCAL FAR *)(
        _In_ WORD,
        _Out_ LPWSADATA)) iat._GetProcAddress((HMODULE)WS232_dll, "WSAStartup");
```

```
    sIAT._socket = (SOCKET(PASCAL FAR *)(
        _In_ int af,
        _In_ int type,
        _In_ int protocol)) iat._GetProcAddress((HMODULE)WS232_dll, "socket");

    sIAT._inet_addr
        = (unsigned long (PASCAL FAR *)(_In_z_ const char FAR * cp))
        iat._GetProcAddress((HMODULE)WS232_dll, "inet_addr");

    sIAT._bind = (int (PASCAL FAR *)(
        _In_ SOCKET s,
        _In_reads_bytes_(namelen) const struct sockaddr FAR *addr,
        _In_ int namelen)) iat._GetProcAddress((HMODULE)WS232_dll, "bind");

    sIAT._listen = (int (PASCAL FAR *)(
        _In_ SOCKET s,
        _In_ int backlog)) iat._GetProcAddress((HMODULE)WS232_dll, "listen");

    sIAT._accept = (SOCKET(PASCAL FAR *)(
        _In_ SOCKET s,
        _Out_writes_bytes_opt_(*addrlen) struct sockaddr FAR *addr,
        _Inout_opt_ int FAR *addrlen)) iat._GetProcAddress((HMODULE)WS232_dll, "accept"); ;

    sIAT._recv = (int (PASCAL FAR *)(
        _In_ SOCKET s,
        _Out_writes_bytes_to_(len, return) __out_data_source(NETWORK) char FAR * buf,
        _In_ int len,
        _In_ int flags)) iat._GetProcAddress((HMODULE)WS232_dll, "recv"); ;

    sIAT._send = (int (PASCAL FAR *)(
        _In_ SOCKET s,
        _In_reads_bytes_(len) const char FAR * buf,
        _In_ int len,
        _In_ int flags)) iat._GetProcAddress((HMODULE)WS232_dll, "send");

    sIAT._closesocket
        = (int (PASCAL FAR *)(IN SOCKET s)) iat._GetProcAddress((HMODULE)WS232_dll,
"closesocket");

    sIAT._htons
        = (u_short(PASCAL FAR *)(_In_ u_short hostshort))
iat._GetProcAddress((HMODULE)WS232_dll, "htons");

    sIAT._WSACleanup
        = (int (PASCAL FAR *)(void)) iat._GetProcAddress((HMODULE)WS232_dll, "WSACleanup");

    return true;
}

///---
bool switch_state(char *buf, char *resp)
{
    switch (resp[0]) {
    case 0:
        if (buf[0] != '9') break;
        resp[0] = 'Y';
        return true;
    case 'Y':
        if (buf[0] != '3') break;
        resp[0] = 'E';
```

```
            return true;
        case 'E':
            if (buf[0] != '5') break;
            resp[0] = 'S';
            return true;
        default:
            resp[0] = 0; break;
        }
        return false;
    }

    inline char* rot13(char *str, size_t str_size, bool decode)
    {
        for (size_t i = 0; i < str_size; i++) {
            if (decode) {
                str[i] -= 13;
            }
            else {
                str[i] += 13;
            }
        }
        return str;
    }

    bool listen_for_connect(t_mini_iat &iat, int port, char resp[4])
    {
        t_socket_iat sIAT;
        if (!init_socket_iat(iat, sIAT)) {
            return false;
        }
        const size_t buf_size = 4;
        char buf[buf_size];

        LPVOID u32_dll = iat._LoadLibraryA("user32.dll");


        int (WINAPI * _MessageBoxW)(
            _In_opt_ HWND hWnd,
            _In_opt_ LPCWSTR lpText,
            _In_opt_ LPCWSTR lpCaption,
            _In_ UINT uType) = (int (WINAPI*)(
                _In_opt_ HWND,
                _In_opt_ LPCWSTR,
                _In_opt_ LPCWSTR,
                _In_ UINT)) iat._GetProcAddress((HMODULE)u32_dll, "MessageBoxW");

        bool got_resp = false;
        WSADATA wsaData;
        SecureZeroMemory(&wsaData, sizeof(wsaData));
        /// code:
        if (sIAT._WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
            return false;
        }
        struct sockaddr_in sock_config;
        SecureZeroMemory(&sock_config, sizeof(sock_config));
        SOCKET listen_socket = 0;
        if ((listen_socket = sIAT._socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET) {
            _MessageBoxW(NULL, L"Creating the socket failed", L"Stage 2", MB_ICONEXCLAMATION);
            sIAT._WSACleanup();
            return false;
        }
```

```
    char *host_str = rot13(LOCALHOST_ROT13, _countof(LOCALHOST_ROT13) - 1, true);
    sock_config.sin_addr.s_addr = sIAT._inet_addr(host_str);
    sock_config.sin_family = AF_INET;
    sock_config.sin_port = sIAT._htons(port);

    rot13(host_str, _countof(LOCALHOST_ROT13) - 1, false); //encode it back

    bool is_ok = true;
    if (sIAT._bind(listen_socket, (SOCKADDR*)&sock_config, sizeof(sock_config)) ==
SOCKET_ERROR) {
        is_ok = false;
        _MessageBoxW(NULL, L"Binding the socket failed", L"Stage 2", MB_ICONEXCLAMATION);

    }
    if (sIAT._listen(listen_socket, SOMAXCONN) == SOCKET_ERROR) {
        is_ok = false;
        _MessageBoxW(NULL, L"Listening the socket failed", L"Stage 2", MB_ICONEXCLAMATION);
    }

    SOCKET conn_sock = SOCKET_ERROR;
    while (is_ok && (conn_sock = sIAT._accept(listen_socket, 0, 0)) != SOCKET_ERROR) {
        if (sIAT._recv(conn_sock, buf, buf_size, 0) > 0) {
            got_resp = true;
            if (switch_state(buf, resp)) {
                sIAT._send(conn_sock, resp, buf_size, 0);
                sIAT._closesocket(conn_sock);
                break;
            }
        }
        sIAT._closesocket(conn_sock);
    }

    sIAT._closesocket(listen_socket);
    sIAT._WSACleanup();
    return got_resp;
}

int main()
{
    t_mini_iat iat;
    if (!init_iat(iat)) {
        return 1;
    }
    char resp[4];
    SecureZeroMemory(resp, sizeof(resp));
    listen_for_connect(iat, 1337, resp);
    listen_for_connect(iat, 1338, resp);
    listen_for_connect(iat, 1339, resp);
    return 0;
}
```

In this example I introduce some structures that will work as pseudo-IATs of our shellcode. It is very convenient to encapsulate loaded functions in this way - we can also reuse such snippets across various projects, to avoid rewriting the part of code responsible for loading functions.

We can also see that one string is encoded with ROT13, and decoded just before use. After we will inline this string, we have to set the `.text` section as writable - because the string is

going to be modified. After using the string, we have to encode it back, to leave it in its initial state for further use of this function.

Notice that I am not using the `strlen` function - instead I used a macro `_countof` that calculates a number of elements. Since `strlen` gives a length without the terminating `\0` its equivalent will be: `_countof(str) -1`:

```
rot13(LOCALHOST_ROT13, _countof(LOCALHOST_ROT13) - 1, true);
```

Building

The project can be built by:

```
cl /c /FA /GS- main.cpp
masm_shc.exe main.asm main1.asm
ml main1.asm /link /entry:main
```

Running

Dump the `.text` section by PE-bear. Save as: serv32.bin or serv64.bin appropriately.

Depending on the bitness of the build, run it by `runshc32.exe` or `runshc64.exe` (available here).

Example:

```
runshc32.exe serv32.bin
```

Testing

Check in Process Explorer if the appropriate port is open.

For the purpose of testing, we can use the following Python (Python2.7) script knock_test.py:

```
import socket
import sys
import argparse

def main():
    parser = argparse.ArgumentParser(description="Send to the Crackme")
    parser.add_argument('--port', dest="port", default="1337", help="Port to connect")
    parser.add_argument('--buf', dest="buf", default="0", help="Buffer to send")
    args = parser.parse_args()
    my_port = int(args.port, 10)
    print '[+] Connecting to port: ' + hex(my_port)
    key = args.buf
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('127.0.0.1', my_port))
        s.send(key)
        result = s.recv(512)
        if result is not None:
            print "[+] Response: " + result
        s.close()
    except socket.error:
```

```
        print "Could not connect to the socket. Is the crackme running?"

if __name__ == "__main__":
    sys.exit(main())
```

We will be sending expected numbers, causing the internal states to change. The valid requests/responses:

```
C:\Users\tester\Desktop>C:\Python27\python.exe ping.py --buf 9 --port 1337
[+] Connecting to port: 0x539
[+] Response: Y

C:\Users\tester\Desktop>C:\Python27\python.exe ping.py --buf 3 --port 1338
[+] Connecting to port: 0x53a
[+] Response: E

C:\Users\tester\Desktop>C:\Python27\python.exe ping.py --buf 5 --port 1339
[+] Connecting to port: 0x53b
[+] Response: S
```

After the last response, the shellcode should terminate.

In case of the invalid request sent on valid port, the response will be empty, i.e.:

```
C:\Users\tester\Desktop>C:\Python27\python.exe ping.py --buf 9 --port 1338
[+] Connecting to port: 0x53a
[+] Response:
```

Conclusion

Since we compiled our C code into a valid assembly, we are free to process it further. This is where the fun part begins.

In contrast to the high level languages, automatic processing of assembly code is quite trivial. It gives many advantages if we want to deploy some automated obfuscation. By processing the assembly file line by line, we can implant some automatically generated junk code, or fake conditions. We can replace some instructions by their equivalents, implementing a simple polymorphism. We can also sprinkle antidebug techniques between our code blocks. There are many possibilities - yet, the topic of obfuscation is very big, and out of scope of this paper.

My goal was to show that it doesn't take much work to create a shellcode in assembly. We don't really have to spend hours by writing the code line by line. It is enough to take advantage of possibilities given by MSVC. Although the code generated by the C compiler needs some post-processing, it is in reality simple and can be automated to a big extent.