Executing Position Independent Shellcode from Object Files in Memory

bruteratel.com/research/feature-update/2021/01/30/OBJEXEC

Chetan Nayak January 30, 2021

Reflective DLL and shellcode injection remain one of the most used techniques for threat actors as well as Red Teamers for post exploitation since the executions happen only in memory and they don't have to drop anything to disk. However, most of the offsec-tools use shellcode injection only for initial access or for exploitation of vulnerable services and processes. Reflective DLLs and C# executables which can be loaded directly into memory are more often the choice of adversaries for post-exploitation tasks, since writing them is a less tedious task than writing assembly code as we have to manage the stack and registers ourselves in assembly. However, what if we can write the shellcode in a higher language like C? In this blog, we will delve into the dark corners of linkers and compilers to write a position independent code in C and extract it's shellcode.

Before we proceed, the whole code mentioned in the blog is uploaded on my github.

A window's compiled executable has various different headers and sections. They usually start with a DOS header, PE header, Optional header and then the sections namely .text, .bss, .idata, .edata, .rdata and so on. The benefit of writing a shellcode in C over a reflective DLL or a C# executable is that we don't have to worry about our PE getting detected due to having a PE header or a DOS header or any obvious strings in memory when you inject the shellcode to a remote process. AMSI doesn't detect shellcodes either and the size of the shellcode will always be much lower than that of a DLL or a C# executable.

While writing shellcode in C, we have to take precaution that our compiled executable has only one section which is the executable section (.text section). PEs have their global variables stored in .bss section, imported DLLs in .idata, exported dlls in .edata. Since we need to have only .text section, we cannot have global variables, imported symbols or static strings in our code. We cannot have char or wchar array based strings in our functions, since char arrays are stored in .rdata section. Our aim here is to write everything in the .text section, extract the opcodes from the .text section and execute them in memory. This means we have to resolve the imports during runtime and we cannot use static libraries. We will have to use byte arrays for strings instead of static char arrays. By default, the linker links the entrypoint of the PE to *mainCRTStartup* which does the job of loading necessary dlls, parsing command line arguments and so on. We will have to change this entrypoint to one of our own function since we don't want to link msvcrt.dll to our executable. If we are focusing on writing x64 shellcode, we have to make sure that our shellcode lands in a 16-byte stack

alignment, unlike x32 shellcode which by default lands up in a 4-byte stack alignment. We will be writing the code which is independent of Microsoft's cl.exe compiler and can be compiled with MingW GCC cross compiler.

So, if we combine everything above, we have to achieve the following:

- 16-byte stack alignment
- Only .text section should exist in the compiled executable
- No independent char array or wchar array strings
- Resolving all imports during runtime
- A linker script to replace *mainCRTStartup* function with our custom entrypoint

In order to make sure that our shellcode is always stack aligned, we will write a small assembly stub which will align the stack and call our C function which would act as our entrypoint. We will convert this assembly code to an object file which we will later link to our C source code. We will write a quick function that extracts the privileges of the current user and prints it on screen and name this C function as *getprivs*. We will add this function as an external function in our assembly code since we won't be writing the *getprivs* function in assembly. We will convert this assembly file to an object file and write all our code in C which does the required task.

```
extern getprivs
global alignstack

segment .text

alignstack:
    push rdi                    ; backup rdi since we will be using this as our main
register
    mov rdi, rsp                ; save stack pointer to rdi
    and rsp, byte -0x10         ; align stack with 16 bytes
    sub rsp, byte +0x20         ; allocate some space for our C function
    call getprivs               ; call the C function
    mov rsp, rdi                ; restore stack pointer
    pop rdi                     ; restore rdi
    ret                         ; return where we left
```

We will name the above asm file as adjuststack.asm and compile it using mingw:

```
nasm -f win64 adjuststack.asm -o objects/adjuststack.o
```

Since we are extracting the privileges of the current user, the exported symbols or WinAPIs that we will require are LoadLibraryA, CloseHandle, GetCurrentProcess from kernel32.dll, OpenProcessToken, GetTokenInformation, LookupPrivilegeNameW from advapi32.dll, and calloc and wprintf from msvcrt.dll. Keep in mind that we will be changing the entrypoint as well and that's why we will need to resolve msvcrt.dll too since we don't want it to be statically linked when creating our object file. We will resolve these exports during runtime by

calculating the ror13 hash for the kernel32.dll and finding the address of LoadLibraryA, then loading the required library using LoadLibraryA and extracting the function pointer for each of the Windows exported Symbols mentioned above. We will not use GetProcAddress since it is one of the most hooked Windows API by AVs and EDRs. We will write our own GetProcAddress function in C which will parse the DLL loaded after LoadLibraryA and extract the Symbol's Pointer. We will typecast each of these 64-bit pointer addresses to our typedefs. Below are the required typecasts for the required symbols.

```
#include "addresshunter.h"
#include <stdio.h>
#include <inttypes.h>

// kernel32.dll exports
typedef HMODULE(WINAPI* LOADLIBRARYA)(LPCSTR);
typedef BOOL(WINAPI* CLOSEHANDLE)(HANDLE);
typedef HANDLE(WINAPI* GETCURRENTPROCESS)();

// advapi32.dll exports
typedef BOOL(WINAPI* OPENPROCESSTOKEN)(HANDLE, DWORD, PHANDLE);
typedef BOOL(WINAPI* GETTOKENINFORMATION)(HANDLE, TOKEN_INFORMATION_CLASS, LPVOID,
DWORD, PDWORD);
typedef BOOL(WINAPI* LOOKUPPRIVILEGENAMEW)(LPCWSTR,  PLUID, LPWSTR, LPDWORD);

// msvcrt.dll exports
typedef int(WINAPI* WPRINTF)(const wchar_t* format, ...);
typedef void*(WINAPI* CALLOC)(size_t num, size_t size);
```

In order the get the address of the symbol from a DLL, we will use the below C function which we will add in *addresshunter.h*:

```
#include <windows.h>
#include <inttypes.h>

#define DEREF( name )*(UINT_PTR *)(name)
#define DEREF_64( name )*(DWORD64 *)(name)
#define DEREF_32( name )*(DWORD *)(name)
#define DEREF_16( name )*(WORD *)(name)
#define DEREF_8( name )*(BYTE *)(name)

#define KERNEL32DLL_HASH 0x6A4ABC5B

//redefine UNICODE_STR struct
typedef struct _UNICODE_STR
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR pBuffer;
} UNICODE_STR, *PUNICODE_STR;

//redefine PEB_LDR_DATA struct
typedef struct _PEB_LDR_DATA
{
    DWORD dwLength;
    DWORD dwInitialized;
    LPVOID lpSsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    LPVOID lpEntryInProgress;
} PEB_LDR_DATA, * PPEB_LDR_DATA;

//redefine LDR_DATA_TABLE_ENTRY struct
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STR FullDllName;
    UNICODE_STR BaseDllName;
    ULONG Flags;
    SHORT LoadCount;
    SHORT TlsIndex;
    LIST_ENTRY HashTableEntry;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

//redefine PEB_FREE_BLOCK struct
typedef struct _PEB_FREE_BLOCK
{
    struct _PEB_FREE_BLOCK * pNext;
```

```
    DWORD dwSize;
} PEB_FREE_BLOCK, * PPEB_FREE_BLOCK;

//redefine PEB struct
typedef struct __PEB
{
    BYTE bInheritedAddressSpace;
    BYTE bReadImageFileExecOptions;
    BYTE bBeingDebugged;
    BYTE bSpareBool;
    LPVOID lpMutant;
    LPVOID lpImageBaseAddress;
    PPEB_LDR_DATA pLdr;
    LPVOID lpProcessParameters;
    LPVOID lpSubSystemData;
    LPVOID lpProcessHeap;
    PRTL_CRITICAL_SECTION pFastPebLock;
    LPVOID lpFastPebLockRoutine;
    LPVOID lpFastPebUnlockRoutine;
    DWORD dwEnvironmentUpdateCount;
    LPVOID lpKernelCallbackTable;
    DWORD dwSystemReserved;
    DWORD dwAtlThunkSListPtr32;
    PPEB_FREE_BLOCK pFreeList;
    DWORD dwTlsExpansionCounter;
    LPVOID lpTlsBitmap;
    DWORD dwTlsBitmapBits[2];
    LPVOID lpReadOnlySharedMemoryBase;
    LPVOID lpReadOnlySharedMemoryHeap;
    LPVOID lpReadOnlyStaticServerData;
    LPVOID lpAnsiCodePageData;
    LPVOID lpOemCodePageData;
    LPVOID lpUnicodeCaseTableData;
    DWORD dwNumberOfProcessors;
    DWORD dwNtGlobalFlag;
    LARGE_INTEGER liCriticalSectionTimeout;
    DWORD dwHeapSegmentReserve;
    DWORD dwHeapSegmentCommit;
    DWORD dwHeapDeCommitTotalFreeThreshold;
    DWORD dwHeapDeCommitFreeBlockThreshold;
    DWORD dwNumberOfHeaps;
    DWORD dwMaximumNumberOfHeaps;
    LPVOID lpProcessHeaps;
    LPVOID lpGdiSharedHandleTable;
    LPVOID lpProcessStarterHelper;
    DWORD dwGdiDCAttributeList;
    LPVOID lpLoaderLock;
    DWORD dwOSMajorVersion;
    DWORD dwOSMinorVersion;
    WORD wOSBuildNumber;
    WORD wOSCSDVersion;
    DWORD dwOSPlatformId;
```

```
    DWORD dwImageSubsystem;
    DWORD dwImageSubsystemMajorVersion;
    DWORD dwImageSubsystemMinorVersion;
    DWORD dwImageProcessAffinityMask;
    DWORD dwGdiHandleBuffer[34];
    LPVOID lpPostProcessInitRoutine;
    LPVOID lpTlsExpansionBitmap;
    DWORD dwTlsExpansionBitmapBits[32];
    DWORD dwSessionId;
    ULARGE_INTEGER liAppCompatFlags;
    ULARGE_INTEGER liAppCompatFlagsUser;
    LPVOID lppShimData;
    LPVOID lpAppCompatInfo;
    UNICODE_STR usCSDVersion;
    LPVOID lpActivationContextData;
    LPVOID lpProcessAssemblyStorageMap;
    LPVOID lpSystemDefaultActivationContextData;
    LPVOID lpSystemAssemblyStorageMap;
    DWORD dwMinimumStackCommit;
} _PEB, * _PPEB;

// main hashing function for ror13
__forceinline DWORD ror13( DWORD d )
{
    return _rotr( d, 13 );
}

__forceinline DWORD hash( char * c )
{
    register DWORD h = 0;
    do
    {
        h = ror13( h );
        h += *c;
    } while( *++c );

    return h;
}

// function to fetch the base address of kernel32.dll from the Process Environment
Block
UINT64 GetKernel32() {
    ULONG_PTR kernel32dll, val1, val2, val3;
    USHORT usCounter;

    // kernel32.dll is at 0x60 offset and __readgsqword is compiler intrinsic,
    // so we don't need to extract it's symbol
    kernel32dll = __readgsqword( 0x60 );

    kernel32dll = (ULONG_PTR)((_PPEB)kernel32dll)->pLdr;
    val1 = (ULONG_PTR)((PPEB_LDR_DATA)kernel32dll)->InMemoryOrderModuleList.Flink;
    while( val1 ) {
```

```
        val2 = (ULONG_PTR)((PLDR_DATA_TABLE_ENTRY)val1)->BaseDllName.pBuffer;
        usCounter = ((PLDR_DATA_TABLE_ENTRY)val1)->BaseDllName.Length;
        val3 = 0;

        //calculate the hash of kernel32.dll
        do {
            val3 = ror13( (DWORD)val3 );
            if( *((BYTE *)val2) >= 'a' )
                val3 += *((BYTE *)val2) - 0x20;
            else
                val3 += *((BYTE *)val2);
            val2++;
        } while( --usCounter );

        // compare the hash kernel32.dll
        if( (DWORD)val3 == KERNEL32DLL_HASH ) {
            //return kernel32.dll if found
            kernel32dll = (ULONG_PTR)((PLDR_DATA_TABLE_ENTRY)val1)->DllBase;
            return kernel32dll;
        }
        val1 = DEREF( val1 );
    }
    return 0;
}

// custom strcmp function since this function will be called by GetSymbolAddress
// which means we have to call strcmp before loading msvcrt.dll
// so we are writing our own my_strcmp so that we don't have to play with egg or
chicken dilemma
int my_strcmp (const char *p1, const char *p2) {
    const unsigned char *s1 = (const unsigned char *) p1;
    const unsigned char *s2 = (const unsigned char *) p2;
    unsigned char c1, c2;
    do {
        c1 = (unsigned char) *s1++;
        c2 = (unsigned char) *s2++;
        if (c1 == '\0') {
            return c1 - c2;
        }
    }
    while (c1 == c2);
    return c1 - c2;
}

UINT64 GetSymbolAddress( HANDLE hModule, LPCSTR lpProcName ) {
    UINT64 dllAddress = (UINT64)hModule,
        symbolAddress = 0,
        exportedAddressTable = 0,
        namePointerTable = 0,
        ordinalTable = 0;

    if( hModule == NULL ) {
```

```
        return 0;
    }

    PIMAGE_NT_HEADERS ntHeaders = NULL;
    PIMAGE_DATA_DIRECTORY dataDirectory = NULL;
    PIMAGE_EXPORT_DIRECTORY exportDirectory = NULL;

    ntHeaders = (PIMAGE_NT_HEADERS)(dllAddress + ((PIMAGE_DOS_HEADER)dllAddress)-
>e_lfanew);
    dataDirectory = (PIMAGE_DATA_DIRECTORY)&ntHeaders->OptionalHeader.DataDirectory[
IMAGE_DIRECTORY_ENTRY_EXPORT ];
    exportDirectory = (PIMAGE_EXPORT_DIRECTORY)( dllAddress + dataDirectory-
>VirtualAddress );

    exportedAddressTable = ( dllAddress + exportDirectory->AddressOfFunctions );
    namePointerTable = ( dllAddress + exportDirectory->AddressOfNames );
    ordinalTable = ( dllAddress + exportDirectory->AddressOfNameOrdinals );

    if (((UINT64)lpProcName & 0xFFFF0000 ) == 0x00000000) {
        exportedAddressTable += ( ( IMAGE_ORDINAL( (UINT64)lpProcName ) -
exportDirectory->Base ) * sizeof(DWORD) );
        symbolAddress = (UINT64)( dllAddress + DEREF_32(exportedAddressTable) );
    }
    else {
        DWORD dwCounter = exportDirectory->NumberOfNames;
        while( dwCounter-- ) {
            char * cpExportedFunctionName = (char *)(dllAddress + DEREF_32(
namePointerTable ));
            if( my_strcmp( cpExportedFunctionName, lpProcName ) == 0 ) {
                exportedAddressTable += ( DEREF_16( ordinalTable ) * sizeof(DWORD) );
                symbolAddress = (UINT64)(dllAddress + DEREF_32( exportedAddressTable
));
                break;
            }
            namePointerTable += sizeof(DWORD);
            ordinalTable += sizeof(WORD);
        }
    }

    return symbolAddress;
}
```

And since we are going to change the entrypoint, we will have to tell the linker as well that we are changing our entrypoint. We will redirect the entrypoint to our assembly label *alignstack* which will align the 16-byte stack for us and then we will call the *getprivs()* function. We will not write any *int main()* or *void main()*, so we will have to make sure that our linker aligns the order of function execution properly. Below is the linker script which will do this.

```
ENTRY(alignstack)
SECTIONS
{
    .text :
    {
        *(.text.alignstack)
        *(.text.getprivs)
    }
}
```

Now that everything is set, we will write our entrypoint function. If you check all the above C code, you will see that we have not used any global variables or any static char array strings. But most of the time we will be forced to use char array strings for formatting or printing the output of our shellcode. In such case, we can do this using char or wchar byte arrays as follows. Writing char strings in byte array makes sure that our strings do not reside in the *.bss* section and the compiler will be forced to include it in the *.text* section of the PE:

```
    CHAR *loadlibrarya_c = "LoadLibraryA"; // will become ->
    CHAR loadlibrarya_c[] = {'L', 'o', 'a', 'd', 'L', 'i', 'b', 'r', 'a', 'r', 'y',
'A', 0};
```

And the final code for extracting the privileges of the current user would look like this:

```
#include "addresshunter.h"
#include <stdio.h>
#include <inttypes.h>

// kernel32.dll exports
typedef HMODULE(WINAPI* LOADLIBRARYA)(LPCSTR);
typedef BOOL(WINAPI* CLOSEHANDLE)(HANDLE);
typedef HANDLE(WINAPI* GETCURRENTPROCESS)();

// advapi32.dll exports
typedef BOOL(WINAPI* OPENPROCESSTOKEN)(HANDLE, DWORD, PHANDLE);
typedef BOOL(WINAPI* GETTOKENINFORMATION)(HANDLE, TOKEN_INFORMATION_CLASS, LPVOID,
DWORD, PDWORD);
typedef BOOL(WINAPI* LOOKUPPRIVILEGENAMEW)(LPCWSTR,  PLUID, LPWSTR, LPDWORD);

// msvcrt.dll exports
typedef int(WINAPI* WPRINTF)(const wchar_t* format, ...);
typedef void*(WINAPI* CALLOC)(size_t num, size_t size);

void getprivs() {
    //dlls to dynamically load during runtime
    UINT64 kernel32dll, msvcrtdll, advapi32dll;
    //symbols to dynamically resolve from dll during runtime
    UINT64 LoadLibraryAFunc, CloseHandleFunc,
        OpenProcessTokenFunc, GetCurrentProcessFunc, GetTokenInformationFunc,
LookupPrivilegeNameWFunc,
        callocFunc, wprintfFunc;

    // kernel32.dll exports
    kernel32dll = GetKernel32();

    CHAR loadlibrarya_c[] = {'L', 'o', 'a', 'd', 'L', 'i', 'b', 'r', 'a', 'r', 'y',
'A', 0};
    LoadLibraryAFunc = GetSymbolAddress((HANDLE)kernel32dll, loadlibrarya_c);

    CHAR getcurrentprocess_c[] = {'G', 'e', 't', 'C', 'u', 'r', 'r', 'e', 'n', 't',
'P', 'r', 'o', 'c', 'e', 's', 's', 0};
    GetCurrentProcessFunc = GetSymbolAddress((HANDLE)kernel32dll,
getcurrentprocess_c);

    CHAR closehandle_c[] = {'C', 'l', 'o', 's', 'e', 'H', 'a', 'n', 'd', 'l', 'e',
0};
    CloseHandleFunc = GetSymbolAddress((HANDLE)kernel32dll, closehandle_c);

    // advapi32.dll exports
    CHAR advapi32_c[] = {'a', 'd', 'v', 'a', 'p', 'i', '3', '2', '.', 'd', 'l', 'l',
0};
    advapi32dll = (UINT64) ((LOADLIBRARYA)LoadLibraryAFunc)(advapi32_c);
    CHAR openprocesstoken_c[] = {'O', 'p', 'e', 'n', 'P', 'r', 'o', 'c', 'e', 's',
's', 'T', 'o', 'k', 'e', 'n', 0};
    OpenProcessTokenFunc = GetSymbolAddress((HANDLE)advapi32dll, openprocesstoken_c);
    CHAR gettokeninformation_c[] = { 'G', 'e', 't', 'T', 'o', 'k', 'e', 'n', 'I',
```

```
'n', 'f', 'o', 'r', 'm', 'a', 't', 'i', 'o', 'n', 0 };
    GetTokenInformationFunc = GetSymbolAddress((HANDLE)advapi32dll,
gettokeninformation_c);
    CHAR lookupprivilegenamew_c[] = {'L', 'o', 'o', 'k', 'u', 'p', 'P', 'r', 'i',
'v', 'i', 'l', 'e', 'g', 'e', 'N', 'a', 'm', 'e', 'W', 0};
    LookupPrivilegeNameWFunc = GetSymbolAddress((HANDLE)advapi32dll,
lookupprivilegenamew_c);

    // msvcrt.dll exports
    CHAR msvcrt_c[] = {'m', 's', 'v', 'c', 'r', 't', '.', 'd', 'l', 'l', 0};
    msvcrtdll = (UINT64) ((LOADLIBRARYA)LoadLibraryAFunc)(msvcrt_c);
    CHAR calloc_c[] = {'c', 'a', 'l', 'l', 'o', 'c', 0};
    callocFunc = GetSymbolAddress((HANDLE)msvcrtdll, calloc_c);
    CHAR wprintf_c[] = {'w', 'p', 'r', 'i', 'n', 't', 'f', 0};
    wprintfFunc = GetSymbolAddress((HANDLE)msvcrtdll, wprintf_c);

    DWORD cbSize = sizeof(TOKEN_ELEVATION), tpSize, length;
    HANDLE hToken = NULL;
    TOKEN_ELEVATION Elevation;
    PTOKEN_PRIVILEGES tPrivs = NULL;
    WCHAR name[256];
    WCHAR priv_enabled[] =  { L'[', L'+', L']', L' ', L'%', L'-', L'5', L'0', L'l',
L's', L' ', L'E', L'n', L'a', L'b', L'l', L'e', L'd', L' ', L'(', L'D', L'e', L'f',
L'a', L'u', L'l', L't', L')', L'\n', 0 };
    WCHAR priv_adjusted[] = { L'[', L'+', L']', L' ', L'%', L'-', L'5', L'0', L'l',
L's', L' ', L'A', L'd', L'j', L'u', L's', L't', L'e', L'd', L'\n', 0 };
    WCHAR priv_disabled[] = { L'[', L'+', L']', L' ', L'%', L'-', L'5', L'0', L'l',
L's', L' ', L'D', L'i', L's', L'a', L'b', L'l', L'e', L'd', L'\n', 0 };
    WCHAR priv_elevated[] =   {L'[', L'+', L']', L' ', L'E', L'l', L'e', L'v', L'a',
L't', L'e', L'd', 0};
    WCHAR priv_restricted[] = {L'[', L'+', L']', L' ', L'R', L'e', L's', L't', L'r',
L'i', L'c', L't', L'e', L'd', 0};

    if (((OPENPROCESSTOKEN)OpenProcessTokenFunc)
(((GETCURRENTPROCESS)GetCurrentProcessFunc)(), TOKEN_QUERY, &hToken)) {
        ((GETTOKENINFORMATION)GetTokenInformationFunc)(hToken, TokenPrivileges,
tPrivs, 0, &tpSize);
        tPrivs = (PTOKEN_PRIVILEGES)((CALLOC)callocFunc)(tpSize+1,
sizeof(TOKEN_PRIVILEGES));

        if (tPrivs) {
            if (((GETTOKENINFORMATION)GetTokenInformationFunc)(hToken,
TokenPrivileges, tPrivs, tpSize, &tpSize)) {
                for(int i=0; i<tPrivs->PrivilegeCount; i++){
                    length=256;
                    ((LOOKUPPRIVILEGENAMEW)LookupPrivilegeNameWFunc)(NULL, &tPrivs-
>Privileges[i].Luid, name, &length);
                    if (tPrivs->Privileges[i].Attributes == 3) {
                        ((WPRINTF)wprintfFunc)(priv_enabled, name);
                    } else if (tPrivs->Privileges[i].Attributes == 2) {
                        ((WPRINTF)wprintfFunc)(priv_adjusted, name);
                    } else if (tPrivs->Privileges[i].Attributes == 0) {
```

```
                        ((WPRINTF)wprintfFunc)(priv_disabled, name);
                    }
                }
            }
        }

        if (((GETTOKENINFORMATION)GetTokenInformationFunc)(hToken, TokenElevation,
&Elevation, sizeof(Elevation), &cbSize)) {
            if (Elevation.TokenIsElevated) {
                ((WPRINTF)wprintfFunc)(priv_elevated);
            } else {
                ((WPRINTF)wprintfFunc)(priv_restricted);
            }
        }
        ((CLOSEHANDLE)CloseHandleFunc)(hToken);
    }
}
```

In order to compile all of the above, we can simply use a makefile as follows:

```
make:
    nasm -f win64 adjuststack.asm -o adjuststack.o
    x86_64-w64-mingw32-gcc getprivs.c -Wall -m64 -ffunction-sections -fno-
asynchronous-unwind-tables -nostdlib -fno-ident -O2 -c -o getprivs.o -Wl,-
Tlinker.ld,--no-seh
    x86_64-w64-mingw32-ld -s adjuststack.o getprivs.o -o getprivs.exe
```

The above makefile does the following:

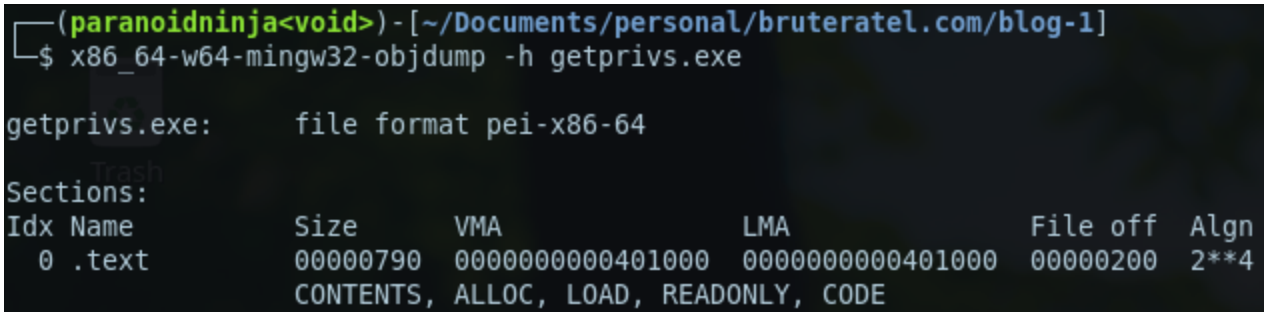
- Create an object file for adjuststack
- Create a 64-bit object file for getprivs.
    - Generate seperate section for each function in the source file and remove unused functions functions during link time.
    - Disable static linking of executables
    - Optimize the size of PE
    - Disable SEH
    - Use argument provided linker script to align function executabe order
- Compile both the above object files and strip all debug symbols and comments

Running the script and using objdump gives us the actual shellcode from the PE.

```
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$ make
```

Extract shellcode from the executable using objdump

You can use objdump to check whether the file has any headers apart from *.text*.



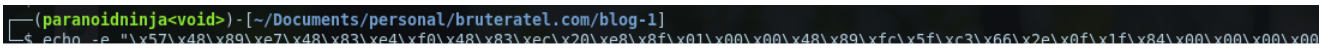
```
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$ x86_64-w64-mingw32-objdump -h getprivs.exe

getprivs.exe:     file format pei-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000790  0000000000401000  0000000000401000  00000200  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Use objdump to list all the sections of the executable

And finally, we can copy the shellcode to a bin file and execute the bin file however we want.



Copy the shellcode to a bin file

For the purpose of this blog, I will use a simple *inc-bin* technique of ASM to execute the shellcode.

```
; compile with
; nasm -f win64 runshellcode.asm -o runshellcode.o
; x86_64-w64-mingw32-ld runshellcode.o -o runshellcode.exe

Global Start

Start:
    incbin "getprivs.bin"
```

The final executable should not be more than 5 kilobyte, which in our case is 4.9kb.

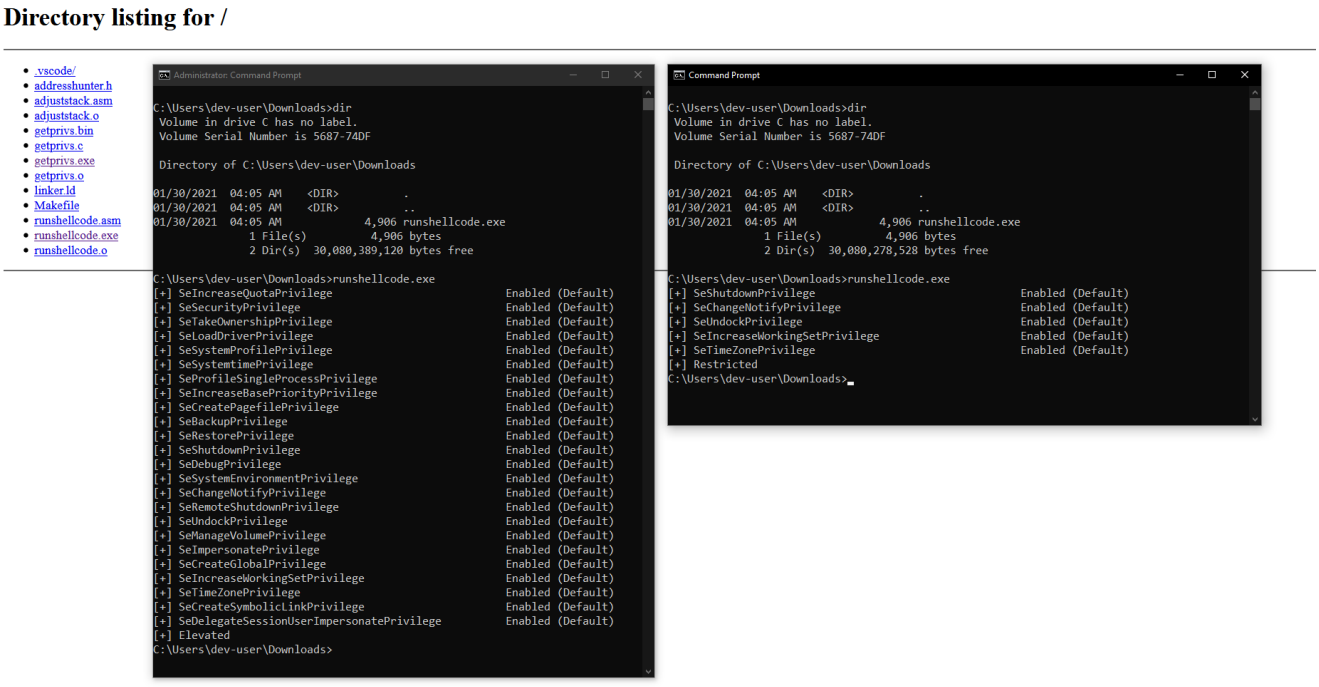
```
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$ cat runshellcode.asm
; compile with
; nasm -f win64 runshellcode.asm -o runshellcode.o
; x86_64-w64-mingw32-ld runshellcode.o -o runshellcode.exe

Global Start

Start:
    incbin "getprivs.bin"
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$ nasm -f win64 runshellcode.asm -o runshellcode.o
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$ x86_64-w64-mingw32-ld runshellcode.o -o runshellcode.exe
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$ ls -al runshellcode.exe
-rwxrwxr-x 1 paranoidninja paranoidninja 4906 Jan 30 17:32 runshellcode.exe
┌──(paranoidninja<void>)-[~/Documents/personal/bruteratel.com/blog-1]
└─$
```

Run the shellcode bin by embedding it with 'incbin' asm

And finally we will execute this on Windows:



As we can see the output differs if we use a privileged cmd prompt and an unprivileged cmd prompt. This brings us to the end of the blog post as to using shellcodes overs C-sharp assembly or Reflective DLLs when injecting custom tools in remote process or even self-injection. We will be adding a feature update to Brute Ratel's next release which will contain executing shellcode from object files within self and remote processes. Stay tuned for more updates.