Module 1

A journey from high level languages, through assembly, to the running process

https://github.com/hasherezade/malware_training_vol1

Creating shellcodes

Shellcode: advantages

- Self-sufficient: easy to inject into other applications
- Small: can fit into a tiny space i.e. section caves
- May be used as a loader: first code injected into an application, that follows to load other modules
- Sometimes (but less often) the full malicious functionality can be implemented as shellcode (i.e. Fobber malware)
- This type of code was popular in the past, virus era: where malware code was added to existing PE files (rather than injected into processes)

Creating shellcode

- 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 conviniences provided by PE format and Windows Loader:
    - No sections
    - No Data Directories (imports, relocations)
    - Only code to provide everything we need…

Creating 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)
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 import table, applied by Windows Loader	Custom: retrieving imports via PEB lookup; no IAT, or simplified

Position-independent code

- In order to create a position-independent code, we must take care that all the addresses that we use are relative to the current instruction pointer address
- A short jump, long jump, call to a local funcion are relative -> we can use them!

```
EBE0          JMP SHORT 0X413BA8
E8EE000000    CALL 0X413BFA
```

- Any address that needs to be relocated (i.e. using of the data from different PE section) breaks the position independence:

```
FF1540414100    CALL DWORD PTR [0X414140]    [KERNEL32.DLL].GetStartupInfoA
391D90A14100    CMP DWORD PTR [0X41A190], EBX
```

Retrieving the Imports

- In order to retrieve the imported functions, we will take advantage of the linklist pointed by PEB

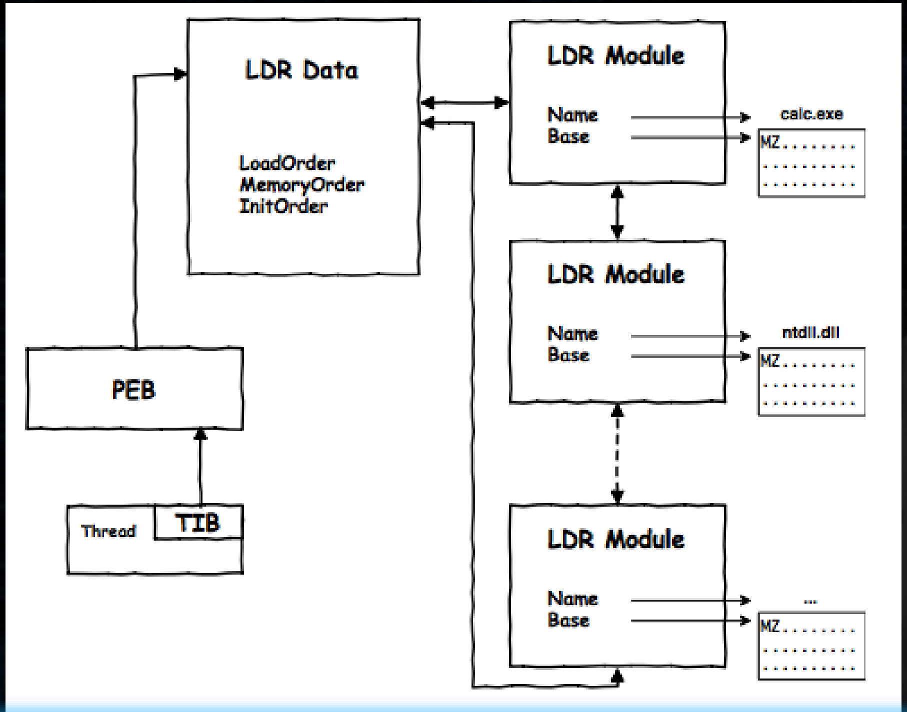


Image from: http://blog.malcom.pl/2017/shellcode-peb-i-adres-bazowy-modulu-kernel32-dll.html

Retrieving the Imports

- In order to retrieve the imported functions, we will take advantage of the linklist pointed by PEB

```
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;
```

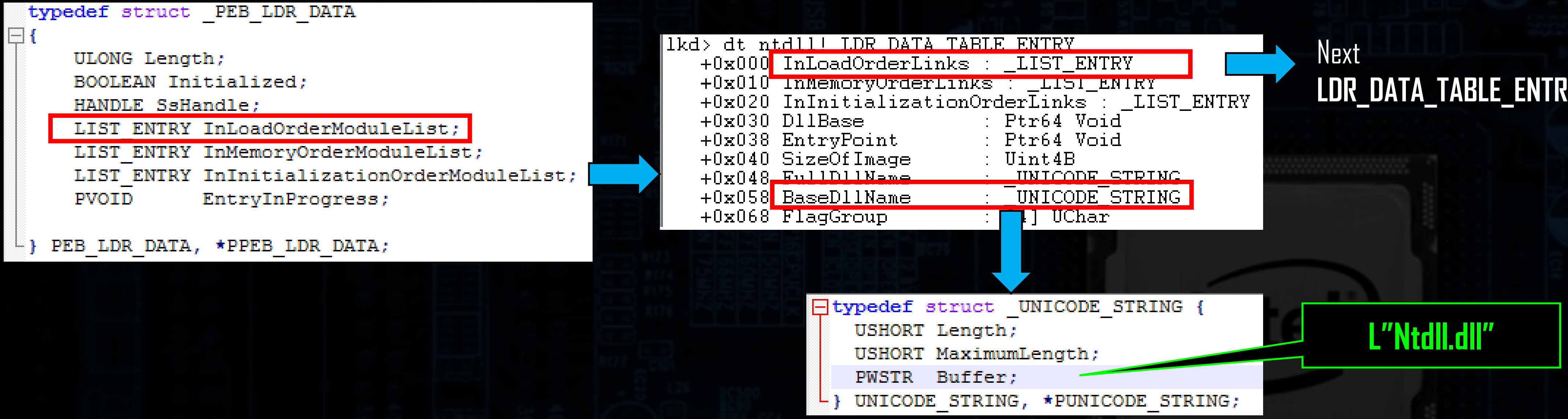
```
typedef struct _PEB
{
     UCHAR InheritedAddressSpace;
     UCHAR ReadImageFileExecOptions;
     UCHAR BeingDebugged;
     UCHAR BitField;
     ULONG ImageUsesLargePages: 1;
     ULONG IsProtectedProcess: 1;
     ULONG IsLegacyProcess: 1;
     ULONG IsImageDynamicallyRelocated: 1;
     ULONG SpareBits: 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;
```

```
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;
```

Retrieving the Imports

- We will process each entry, searching for the DLL that we need…



Retrieving the Imports

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

Creating shellcode: assembly

- We can use YASM for shellcodes written in pure assembly:

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

- We will not use a linker, which means:
  - we need to fill imports by ourselves
  - we need to take care of relocations – or make the code position-independent

Creating shellcode: C

- We can use a C compiler to generate assembly:

```
Cl /c /FA <file_name>.cpp
```

- ...that we will refactor to our shellcode, and compile by masm:

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

- it will generate a PE: we will cut out the code section, that is our shellcode
- The key is the refactoring! We need to follow all the principles of building shellcodes...

Creating shellcode: C

- Use the given template, and refactor the application in C into a valid shellcode, by following the steps…

```
Exercise time...
```

Further readings...

- From a C project, through assembly, to shellcode:
  - https://vxug.fakedoma.in/papers/VXUG/Exclusive/FromaCprojectthroughassemblytoshellcodeHasherezade.pdf