

Module 1

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

https://github.com/hasherezade/malware_training_voll

Running executables: process

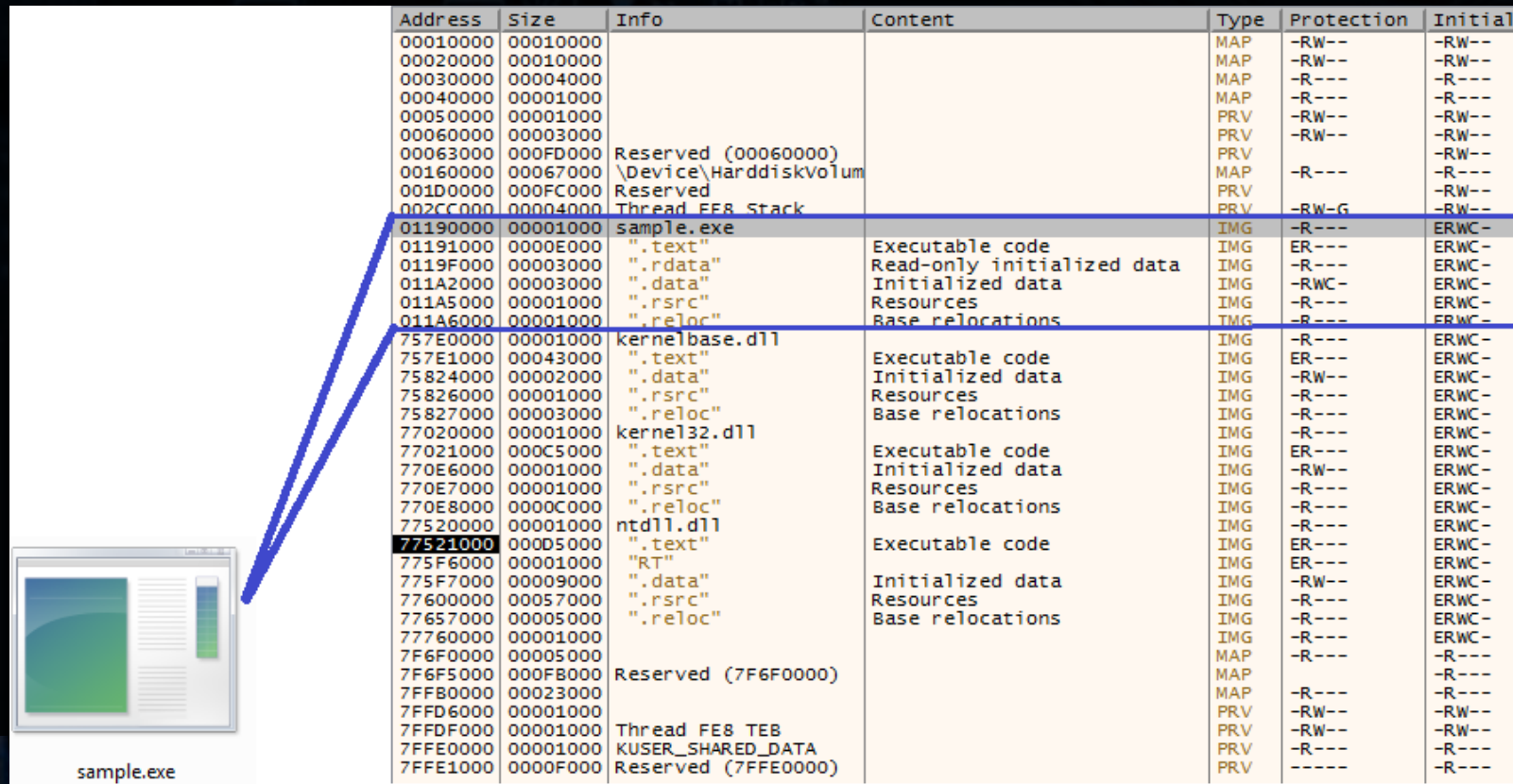


Process: basics



Process

- When we run an EXE file, the system creates a Process



The diagram illustrates the mapping of memory segments from a running process (sample.exe) to a memory dump table. A blue arrow points from the 'sample.exe' icon to the first row of the table (Address 01190000). Another blue arrow points from the 'sample.exe' icon to the 'sample.exe' row in the table. A third blue arrow points from the 'sample.exe' icon to the 'kernelbase.dll' row in the table. A fourth blue arrow points from the 'sample.exe' icon to the 'ntdll.dll' row in the table. A fifth blue arrow points from the 'sample.exe' icon to the '77521000' row in the table.

Address	Size	Info	Content	Type	Protection	Initial
00010000	00010000			MAP	-RW--	-RW--
00020000	00010000			MAP	-RW--	-RW--
00030000	00004000			MAP	-R---	-R---
00040000	00001000			MAP	-R---	-R---
00050000	00001000			PRV	-RW--	-RW--
00060000	00003000			PRV	-RW--	-RW--
00063000	000FD000	Reserved (00060000)		PRV	-RW--	-RW--
00160000	00067000	\Device\HarddiskVolume		MAP	-R---	-R---
001D0000	000FC000	Reserved		PRV	-RW--	-RW--
002C0000	00004000	Thread FFB Stack		PRV	-RW-G	-RW--
01190000	00001000	sample.exe		IMG	-R---	ERWC-
01191000	0000E000	".text"	Executable code	IMG	ER---	ERWC-
0119F000	00003000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
011A2000	00003000	".data"	Initialized data	IMG	-RWC-	ERWC-
011A5000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
011A6000	00001000	".reloc"	Base relocations	IMG	-R---	ERWC-
757E0000	00001000	kernelbase.dll		IMG	-R---	ERWC-
757E1000	00043000	".text"	Executable code	IMG	ER---	ERWC-
75824000	00002000	".data"	Initialized data	IMG	-RW--	ERWC-
75826000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
75827000	00003000	".reloc"	Base relocations	IMG	-R---	ERWC-
77020000	00001000	kernel32.dll		IMG	-R---	ERWC-
77021000	000C5000	".text"	Executable code	IMG	ER---	ERWC-
770E6000	00001000	".data"	Initialized data	IMG	-RW--	ERWC-
770E7000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
770E8000	0000C000	".reloc"	Base relocations	IMG	-R---	ERWC-
77520000	00001000	ntdll.dll		IMG	-R---	ERWC-
77521000	00005000	".text"	Executable code	IMG	ER---	ERWC-
775F6000	00001000	".rt"		IMG	ER---	ERWC-
775F7000	00009000	".data"	Initialized data	IMG	-RW--	ERWC-
77600000	00057000	".rsrc"	Resources	IMG	-R---	ERWC-
77657000	00005000	".reloc"	Base relocations	IMG	-R---	ERWC-
77760000	00001000			IMG	-R---	ERWC-
7F6F0000	00005000			MAP	-R---	-R---
7F6F5000	000FB000	Reserved (7F6F0000)		MAP	-R---	-R---
7FFB0000	00023000			MAP	-R---	-R---
7FFD6000	00001000			PRV	-RW--	-RW--
7FFDF000	00001000	Thread FFB TEB		PRV	-RW--	-RW--
7FFE0000	00001000	KUSER_SHARED_DATA		PRV	-R---	-R---
7FFE1000	0000F000	Reserved (7FFE0000)		PRV	-----	-R---

Process

- A process is a container for all the resources that the application needs to run
- A **process** by itself doesn't run code: **threads** execute it
- Each process has its own, **private address space**, that is independent from other processes (different processes may have different memory content at the same addresses)
- Has its own **access token**, defining its security context

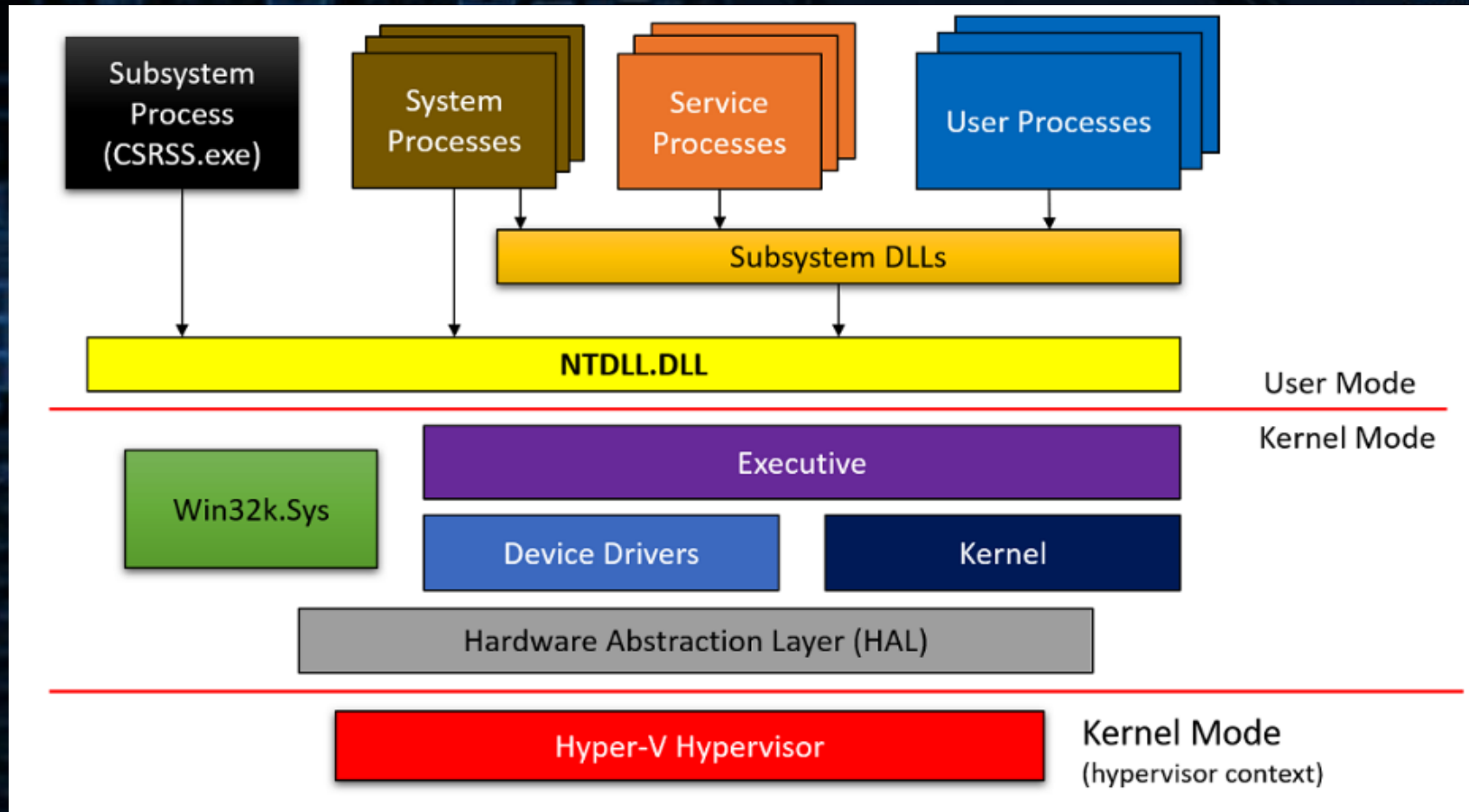


Process

- Types of processes on Windows:
 - System process
 - Subsystem process
 - Service
 - **User processes** (our applications)



Processes on Windows



From: „Windows Kernel Programming“ by Pavel Yosifovich

Process

- A process is identified by its PID (Process ID)
 - unique throughout the system at the time of running
 - after the process terminates, its PID may be reused by a new process
- Each process has one or more threads. They are identified by Thread IDs.
 - Thread IDs, same as process IDs, are unique throughout the system
 - After the thread terminates, its ID may be reused
- Processes may access each other (via handles), if their security context allows it

```
HANDLE OpenProcess(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    DWORD dwProcessId // <- The Process ID  
) ;
```

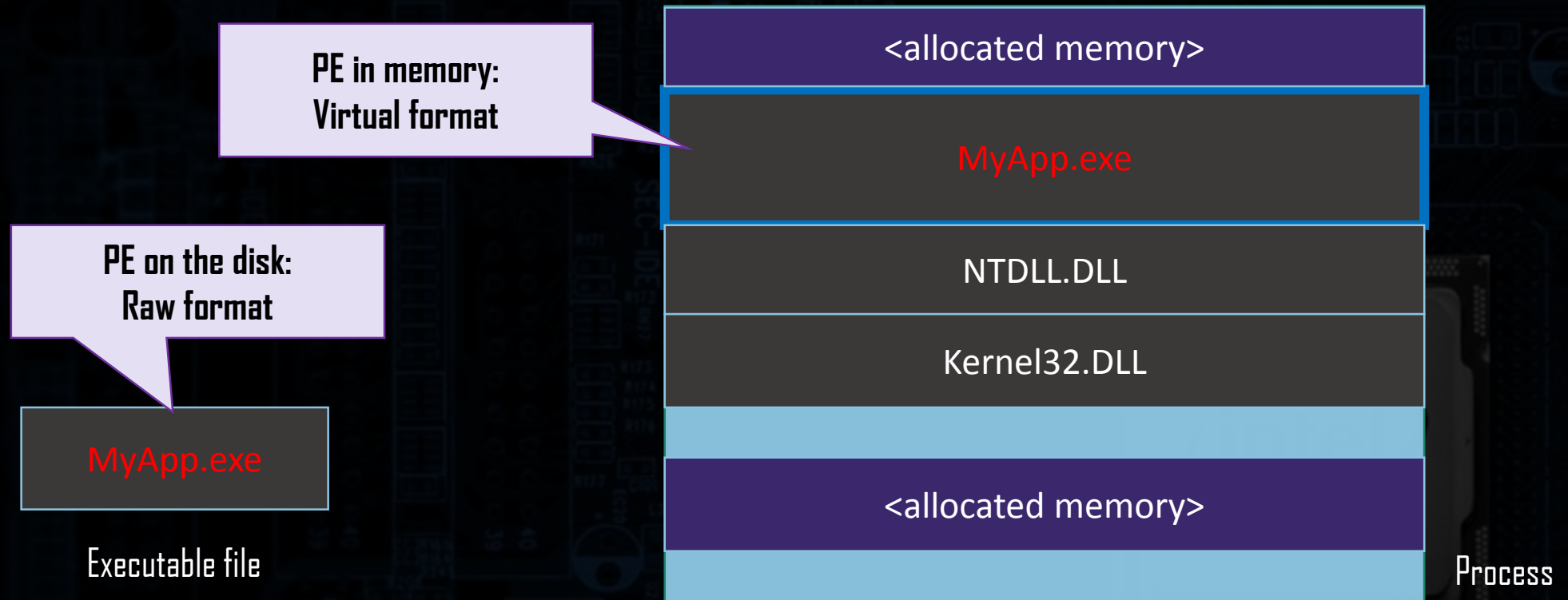

Process

- Process contains:
 - Mapped **PE images** (the main EXE + dependencies: DLLs with needed imports)
 - The **workingset** (all the memory that is used during its execution)
 - **Threads**: at least one (structures for execution of the code)
 - Open **Handles** (managing access to needed objects: i.e. Files, Mutexes, Events)
 - Access **Tokens** (representing security information, and specifying privileges of the process and threads)



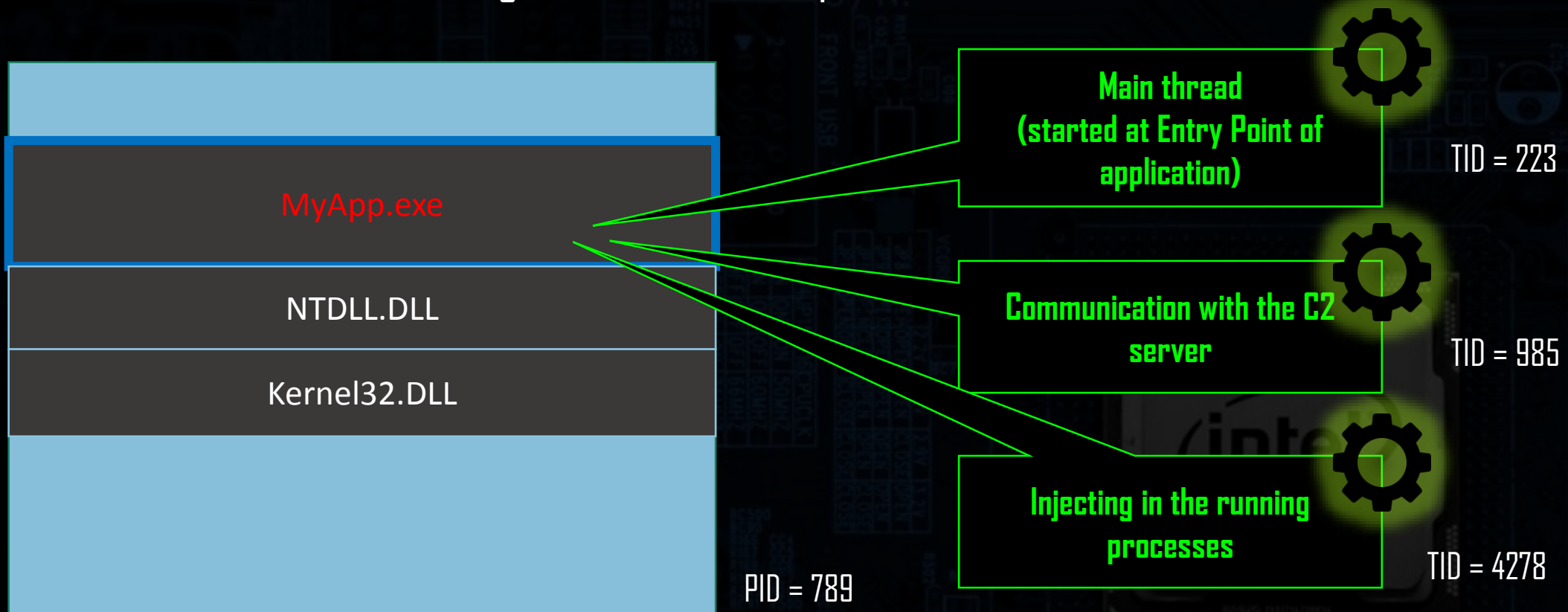
Process

- Contains PE files in a virtual format



Process

- Contains thread(s) running the code – example:



The background of the image is a dark, blue-tinted photograph of a computer circuit board. A large, square Intel processor is visible in the lower right quadrant, with the 'intel' logo clearly printed on its surface. The board is densely packed with various electronic components, including capacitors, resistors, and integrated circuits. Faint white text and markings are visible on the board, such as 'W83877F' at the top, 'S/N:' in the center, and 'SEC-IDE' on the left. The overall aesthetic is technical and high-tech.

Process initialization

Process Initialization

- What happens when we create a process?

```
BOOL CreateProcess(  
    LPCSTR          lpApplicationName,  
    LPSTR           lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL            bInheritHandles,  
    DWORD           dwCreationFlags,  
    LPVOID          lpEnvironment,  
    LPCSTR          lpCurrentDirectory,  
    LPSTARTUPINFOA   lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
)
```

Process Initialization

1. Create a new process object and allocation of the memory
2. Map NTDLL.dll and the initial EXE into the memory (MEM_IMAGE)
3. Create a first thread and allocate a space for it
4. Resume the first thread: `NTDLL.LdrpInitialize` function is called
5. `NTDLL.LdrpInitialization` function:
 - Load all imported DLLs -> run each's `DllMain` with `DLL_PROCESS_ATTACH`
 - Call `Kerne132.BaseProcessStart`
6. `Kerne132.BaseProcessStart`: calls initial EXE's `Entry Point`

Process Initialization

Windows Loader CreateProcess

- Creates process and allocates a virtual memory for its use
- Loads the initial EXE and NDTLL.DLL
- Creates a first thread and the stack for its use

Windows Loader LdrpInitialize

- Called when the first thread resumes
- Goes through the Import Table, loads all required DLLs, and initializes them (calls DllMain with DLL_PROCESS_ATTACH)

Windows Loader BaseProcessStart

- Call Entry Point of the original application

The run EXE Entry Point

- Execute the code at the Entry Point

Process Initialization

Windows Loader CreateProcess

- Creates process and allocates a virtual memory for its use
- Loads the initial EXE and NDTLL.DLL
- Creates a first thread and the stack for its use

Windows Loader LdrpInitialize

- Called when the first thread resumes
- Goes through the Import Table, loads all required DLLs, and initializes them (calls DllMain with DLL_PROCESS_ATTACH)

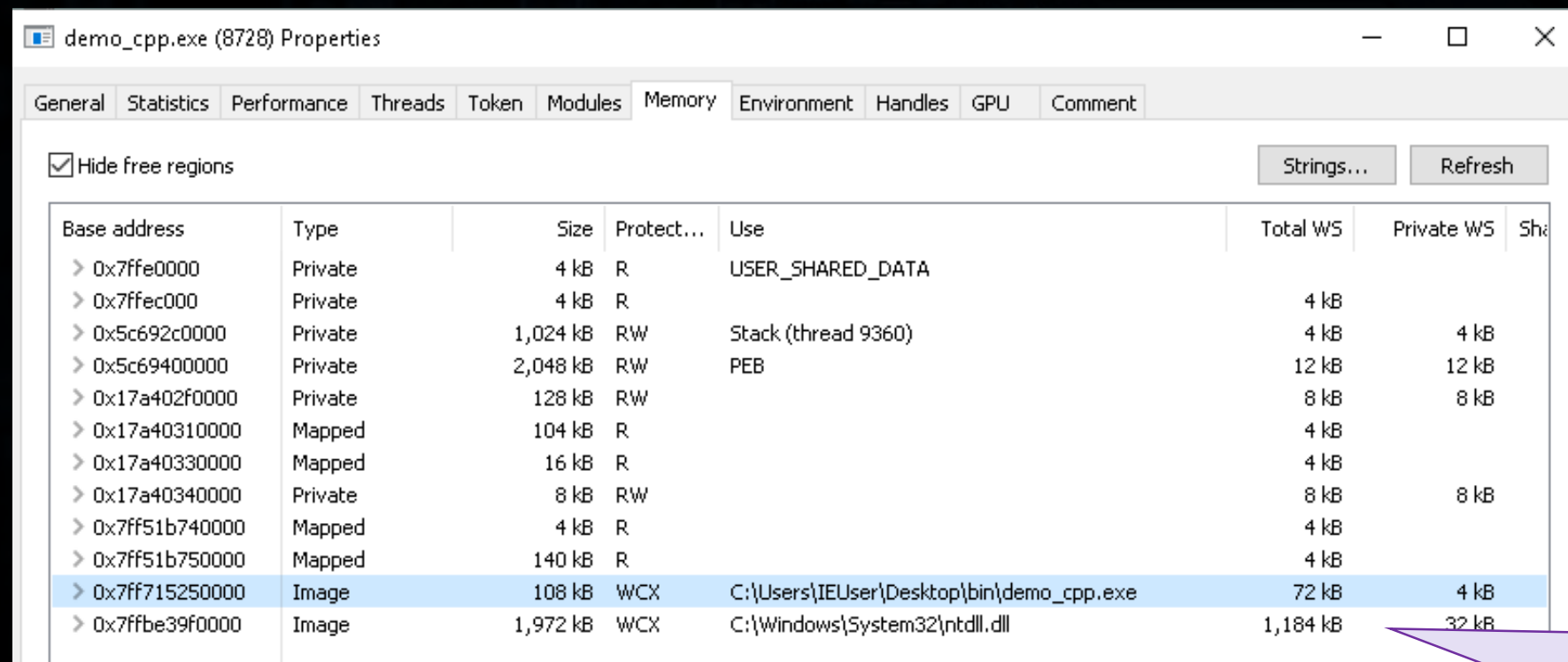
Windows Loader BaseProcessStart

- Call Entry Point of the original application

The run EXE Entry Point

- Execute the code at the Entry Point

Process Initialization



demo_cpp.exe (8728) Properties

General Statistics Performance Threads Token Modules **Memory** Environment Handles GPU Comment

☒ Hide free regions

Strings... Refresh

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Sha
> 0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA			
> 0x7ffec000	Private	4 kB	R		4 kB		
> 0x5c692c0000	Private	1,024 kB	RW	Stack (thread 9360)	4 kB	4 kB	
> 0x5c69400000	Private	2,048 kB	RW	PEB	12 kB	12 kB	
> 0x17a402f0000	Private	128 kB	RW		8 kB	8 kB	
> 0x17a40310000	Mapped	104 kB	R		4 kB		
> 0x17a40330000	Mapped	16 kB	R		4 kB		
> 0x17a40340000	Private	8 kB	RW		8 kB	8 kB	
> 0x7ff51b740000	Mapped	4 kB	R		4 kB		
> 0x7ff51b750000	Mapped	140 kB	R		4 kB		
> 0x7ff715250000	Image	108 kB	WCX	C:\Users\IEUser\Desktop\bin\demo_cpp.exe	72 kB	4 kB	
> 0x7ffbe39f0000	Image	1,972 kB	WCX	C:\Windows\System32\ntdll.dll	1,184 kB	32 kB	

Before the first thread is run, only:

- the main EXE
- NTDLL.DLL

are mapped

A process created in a suspended mode – 64 bit example (viewed by Process Hacker)

Process Initialization

- Notice that if we create a process as **suspended**, only the first part of the initialization process was run...
- This is important for **Process Hollowing**, that we will review in details later...



A close-up, high-resolution photograph of a computer motherboard, specifically focusing on the central processing unit (CPU) area. The image is dark and has a blueish tint. In the lower right, a large, square Intel processor is visible, with the 'intel' logo clearly printed on its surface. The processor is mounted on a complex network of circuitry, including various capacitors, resistors, and other integrated circuits. Labels like 'W83877F' and 'CE' are visible in the upper right. The word 'Threads' is superimposed in a large, white, sans-serif font across the center of the image. The overall composition is technical and detailed, highlighting the intricate design of modern computer hardware.

Threads

Thread

- Thread is an entity responsible for executing the code

Main thread
(started at Entry Point of
application)

TID = 223

```
EAX 76653C33 <kernel32.BaseThreadInitThunk>
EBX 7FFD4000
ECX 00000000
EDX 00413A84 <remcos.EntryPoint>
EBP 0012FF94
ESP 0012FF8C "E<ev"
ESI 00000000
EDI 00000000

EIP 00413A84 <remcos.EntryPoint>

EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

GS 0000 FS 0038
ES 0023 DS 0023
CS 0018 SS 0023
```

```
00413A84 push ebp
00413A85 mov ebp,esp
00413A87 push FFFFFFFF
00413A89 push remcos.415F08
00413A8E push <JMP.&_except_handler3>
00413A93 mov eax,dword ptr FS:[0]
00413A99 push eax
00413A9A mov dword ptr FS:[0],esp
00413AA1 sub esp,68
00413AA4 push ebx
00413AA5 push esi
00413AA6 push edi
00413AA7 mov dword ptr SS:[ebp-18],esp
00413AAA xor ebx,ebx
00413AAC mov dword ptr SS:[ebp-4],ebx
00413AAF push 2
00413AB1 call dword ptr DS:[&_set_app_type]
00413AB7 pop ecx
00413AB8 or dword ptr DS:[41B144],FFFFFFFF
```

MyApp.exe

Thread

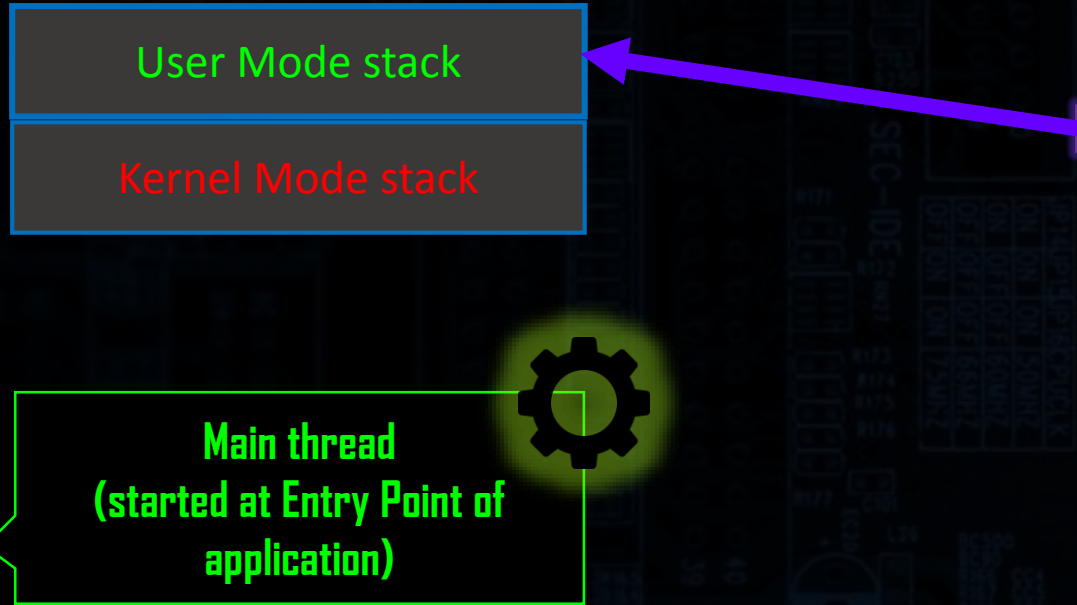
- A thread contains: Context (state of the processor), 2 stacks, TLS (Thread Local Storage), may also has its own security token

User Mode stack

Kernel Mode stack

Main thread
(started at Entry Point of
application)

TID = 223



```
EAX 76653C33 <kernel32.BaseThreadInitThunk>
EBX 7FFD4000
ECX 00000000
EDX 00413A84 <remcos.EntryPoint>
ESP 0012FF8C "E<ev"
ESI 00000000
EDI 00000000

EIP 00413A84 <remcos.EntryPoint>

EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

GS 0000 FS 003B
ES 0023 DS 0023
CS 001B SS 0023
```

Context

Thread Management

- **Threads** are executed by the **processor**, and managed by the **Operating System** (kernel mode):
 - Scheduler: a kernel mode controller, that decides which thread gets to run for how long and performing the context switch
- **Additionally**, Windows (only 64-bit) implements also User Mode Scheduling (**UMS**). It is it an optimization to make the operation of thread switching less resource-consuming. UMS threads differ from classic threads. They can switch context between themselves in user mode, while from the kernel perspective, it looks like one thread is running. Due to this, concurrent UMS Threads cannot run on multiple processors.

Thread Context

- Context switching:
 - When the processor is switched to another thread, first its context is saved
 - The thread context is a state of the processor when it was run the last time before the switch (saved snapshot with all the registers)
 - stack space is used to save off current state of thread when context switched
 - WindowsAPI allows to retrieve the thread context (but first we need to `SuspendThread`):

```
BOOL GetThreadContext(  
    HANDLE    hThread,  
    LPCONTEXT lpContext  
);
```



Thread Context

- Example

Main thread
(started at Entry Point of
application)

TID = 223

```
EAX 76653C33 <kernel32.BaseThreadInitThunk>
EBX 7FFD4000
ECX 00000000
EDX 00413A84 <remcos.EntryPoint>
EBP 0012FF94
ESP 0012FF8C "E<ev"
ESI 00000000
EDI 00000000

EIP 00413A84 <remcos.EntryPoint>

EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

GS 0000 FS 0038
ES 0023 DS 0023
CS 0018 SS 0023
```

```
00413A84 push ebp
00413A85 mov ebp,esp
00413A87 push FFFFFFFF
00413A89 push remcos.415F08
00413A8E push <JMP.&_except_handler3>
00413A93 mov eax,dword ptr FS:[0]
00413A99 push eax
00413A9A mov dword ptr FS:[0],esp
00413AA1 sub esp,68
00413AA4 push ebx
00413AA5 push esi
00413AA6 push edi
00413AA7 mov dword ptr SS:[ebp-18],esp
00413AAA xor ebx,ebx
00413AAC mov dword ptr SS:[ebp-4],ebx
00413AAF push 2
00413AB1 call dword ptr DS:[&_set_app_type]
00413AB7 pop ecx
00413AB8 or dword ptr DS:[41B144],FFFFFFFF
```

MyApp.exe

ПРОЦЕСС, ПЕВ, ТЕВ...



Structures for Process Management

- Process is managed by the Operating System
- To manage the process, Windows uses the following structures:
 - EPROCESS, KPROCESS, ETHREAD, KTHREAD, PEB, TEB...



Structures for Process Management

- **EPROCESS** – the basic kernel-mode structure representing a process
 - Contains a linklist of all the threads belonging to the process
 - Contains a pointer to the **PEB (Process Environment Block)** that is available from usermode
- **ETHREAD** - the basic kernel-mode structure representing a thread
 - Contains a pointer to KTHREAD
 - Links to the **TEB (Thread Environment Block)** that is available from usermode



Obtaining PEB

```
typedef struct _EPROCESS
{
    KPROCESS Pcb;
    EX_PUSH_LOCK ProcessLock;
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER ExitTime;
    EX_RUNDOWN_REF RundownProtect;
    PVOID UniqueProcessId;
    LIST_ENTRY ActiveProcessLinks;
    [...]
    ULONG ActiveThreads;
    ULONG ImagePathHash;
    ULONG DefaultHardErrorProcessing;
    LONG LastThreadExitStatus;
    PPEB Peb;
    EX_FAST_REF PrefetchTrace;
    [...]
    UCHAR PriorityClass;
    MM_AVL_TABLE VadRoot;
    ULONG Cookie;
    ALPC_PROCESS_CONTEXT AlpcContext;
} EPROCESS, *PEPROCESS;
```

Kernel Mode

```
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 _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;
```

User Mode

Obtaining TEB

```
! ETHREAD
Tcb : KTHREAD
CreateTime : _LARGE_INTEGER
ExitTime : _LARGE_INTEGER
KeyedWaitChain : _LIST_ENTRY
PostBlockList : _LIST_ENTRY
ForwardLinkShadow : Ptr64 Void
StartAddress : Ptr64 Void
TerminationPort : Ptr64 _TERMINATION_PORT
ReaperLink : Ptr64 _ETHREAD
KeyedWaitValue : Ptr64 Void
ActiveTimerListLock : UInt8B
ActiveTimerListHead : _LIST_ENTRY
Cid : _CLIENT_ID
KeyedWaitSemaphore : _KSEMAPHORE
AlpcWaitSemaphore : _KSEMAPHORE
ClientSecurity : _PS_CLIENT_SECURITY_CONTEXT
```

Kernel Mode

```
_KTHREAD
Header : _DISPATCHER_HEADER
SListFaultAddress : Ptr64 Void
QuantumTarget : UInt8B
InitialStack : Ptr64 Void
StackLimit : Ptr64 Void
StackBase : Ptr64 Void
WaitStatus : Int8B
WaitBlockList : Ptr64 _KWAIT_BLOCK
WaitListEntry : _LIST_ENTRY
SwapListEntry : _SINGLE_LIST_ENTRY
Queue : Ptr64 _DISPATCHER_HEADER
Teb : Ptr64 Void
RelativeTimerBias : UInt8B
Timer : _KTIMER
WaitBlock : [4] _KWAIT_BLOCK
WaitBlockFill14 : [20] UChar
ContextSwitches : UInt4B
WaitBlockFill15 : [68] UChar
State : UChar
```

Via registry:
FS (32 bit)
GS (64 bit)

```
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;
```

User Mode

PEB and TEB

- We can see PEB and TEB(s) mapped inside the process space (usually towards the end of the addresses)

77359000	00001000	".reloc"	Base relocations	IMG	-R---	ERWC-
77370000	00001000	nsi.dll		IMG	-R---	ERWC-
77371000	00002000	".text"	Executable code	IMG	ER---	ERWC-
77373000	00001000	".data"	Initialized data	IMG	-RWC-	ERWC-
77374000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
77375000	00001000	".reloc"	Base relocations	IMG	-R---	ERWC-
77440000	00001000			IMG	-R---	ERWC-
7F6F0000	00005000			MAP	-R---	-R---
7F6F5000	000FB000	Reserved (7F6F0000)		MAP	-R---	-R---
7FEB0000	00023000			MAP	-R---	-R---
7FFD6000	00001000	PEB		PRV	-RW--	-RW--
7FFDF000	00001000	Thread 87C TEB		PRV	-RW--	-RW--
7FFE0000	00001000	KUSER_SHARED_DATA		PRV	-R---	-R---
7FFE1000	0000F000	Reserved (7FFE0000)		PRV	-----	-R---

Exercise

- Following the given instructions, walk through the PEB and TEB using WinDbg. Familiarize yourself with the fields.