Reverse Engineering and
Malware Analysis Fundamentals

WinAPI & Processes

Adam Podlosky
@apodlosky

Objectives

- Introduce Microsoft Windows API
  - Types of APIs
  - Types of Strings
- Why WinAPI knowledge is relevant to malware analysis
- Unpacking PEs
  - Dumping an executing process back to a PE

Windows API

- Typically referred to as “WinAPI” or “Win32 API”
  - C language interface to operating system
- This is your bible:

  https://docs.microsoft.com/en-us/windows/desktop/api/index
- Newer applications use the Windows Runtime (WinRT)
  - C++ object-oriented interface
  - Available in Windows 8 and newer

File Manipulation APIs

- CopyFileA, CopyFileW - copy
  - CopyFileExA, CopyFileEx
- DeleteFileA, DeleteFileW - unlink
- GetFileSize, GetFileSizeEx - stat
- MoveFileA, MoveFileW - rename
  - MoveFileExA, MoveFileExW
- LockFile, UnlockFile - lock, unlock

Directory Manipulation APIs

- CreateDirectoryA, CreateDirectoryW - mkdir
- RemoveDirectoryA, RemoveDirectoryW - unlink
- FindFirstFileA, FindFirstFileW - opendir/scandir
  - FileFirstFileExA, FindFirstFileExW
- FindNextFile - readdir
- FindFileClose

Basic File I/O APIs

- `CreateFileA, CreateFileW` - open
- `ReadFile, ReadFileEx` - read
- `WriteFile, WriteFileEx` - write
- `SetFilePointer` - seek
- `FlushFileBuffers` - sync
- `CloseHandle` - close
- Many more for async./scattered/transacted I/O, memory mapping

Heap Memory APIs

- `HeapAlloc - malloc`
- `HeapReAlloc - realloc`
- `HeapFree - free`
- `HeapCreate & HeapDestroy`
- `HeapSize`
- Others for setting options, locking, validation...

Legacy Memory APIs

- `LocalAlloc, LocalReAlloc, LocalFree, LocalLock, LocalUnlock, LocalSize`
- `GlobalAlloc, GlobalReAlloc, GlobalFree, GlobalLock, GlobalUnlock, GlobalSize`
- Left-over from 16-bit Windows, for compatibility
  - Still needed for specific purposes
  - Some (poorly written) malware still use these

Memory Page APIs

- `VirtualAlloc, VirtualAllocEx - brk`
- `VirtualProtect, VirtualProtectEx - mprotect`
- `VirtualFree, VirtualFreeEx`
- Handling pages of virtual memory
  - NOT for allocating and freeing small buffers, strings, etc.
- Often used by malware for unpacking and injection

Process and Thread APIs

- CreateProcessA, CreateProcessW - fork (kinda)
- OpenProcess
- TerminateProcess - kill w/SIG_KILL
- CreateThread - pthread_create
- OpenThread
- TerminateThread - pthread_cancel

API Weirdness

- `*A or *W, *Ex`
  - Sometimes all of the above?
- e.g. VirtualAlloc variants
  - `VirtualAlloc, VirtualAllocEx,`
  - `VirtualAllocFromApp,`
  - `VirtualAlloc2, VirtualAlloc2FromApp,`
  - `VirtualAllocExNuma`

So Many, Many More APIs

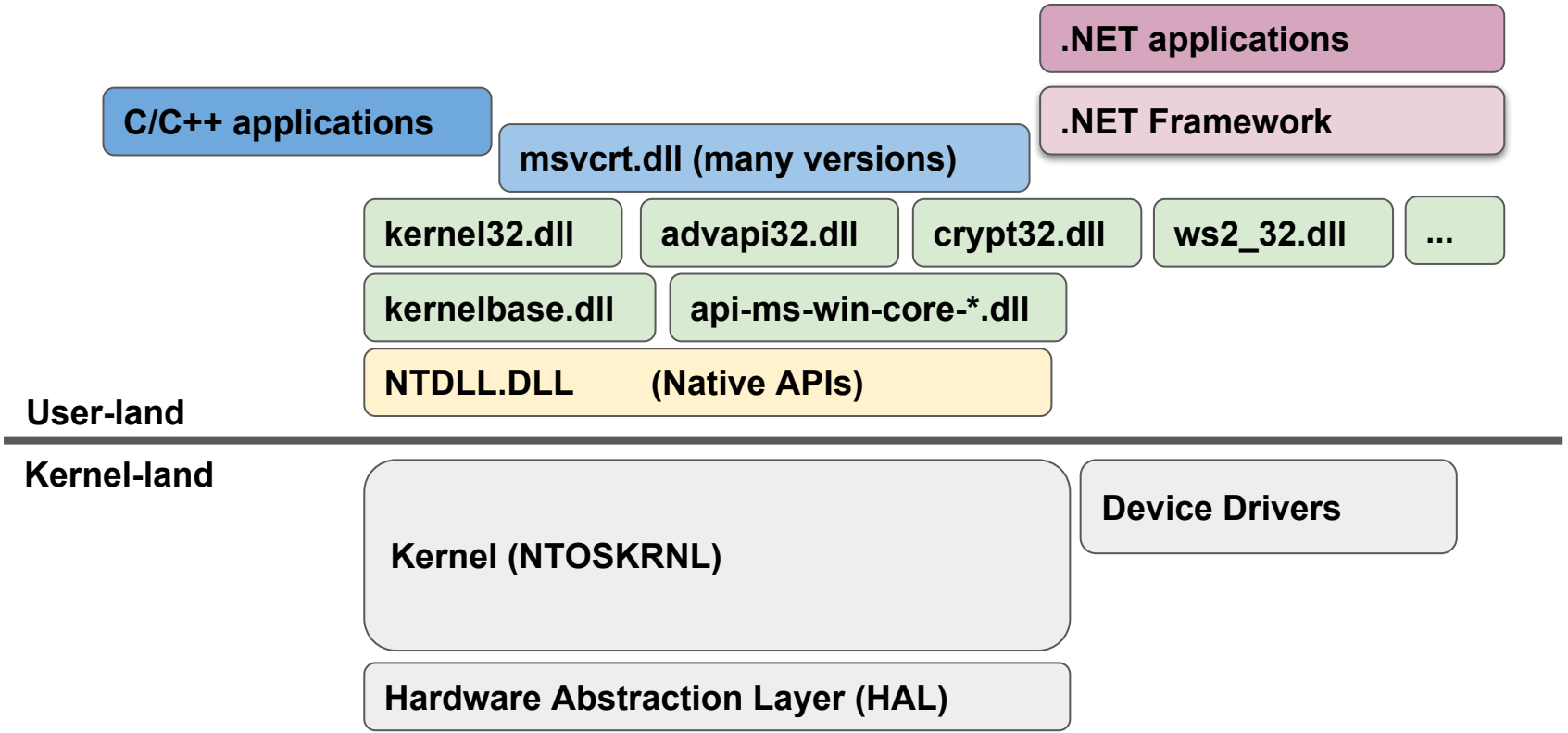
- Graphical interfaces
- Network communication
- Process and thread synchronization
- Cryptography, services, registry, system management, time
- Again, refer to:

  https://docs.microsoft.com/en-us/windows/desktop/api/index

System Calls and the Native API

- 99% of WinAPI functions do NOT call into the kernel directly
  - Unlike Linux; open, read, etc. are syscalls
- Instead, **WinAPI functions call Native API functions**
- Native API lives in NTDLL.DLL, mostly prefixed with Nt*/Zw*
  - Undocumented, unsupported, no compatibility
  - May change between Windows versions, don’t use
  - See https://www.reactos.org/ & http://undocumented.ntinternals.net

Layers of DLLs and APIs



Open a File

1. `CreateFileA("C:\foo.txt", GENERIC_READ, ...);`
2. `CreateFileW(L"C:\foo.txt", GENERIC_READ, ...);`
   a. Convert DOS path to NT path, e.g.
      `\\?\Volume{0FAF43C1-2AED-4824-B2D4-2339C14E93A}\foo.txt`
3. `NtCreateFile(PUNICODE_STRING *FullPath, ...)`
4. Enter kernel-mode
   a. Magic happens
5. Returns a HANDLE to the file

Create a Process

1. **CreateProcessA**(NULL, "notepad.exe", ...);
2. **CreateProcessInternalA**(...)
3. **CreateProcessInternalW**(...)
4. **NtCreateProcess** / **NtCreateProcessEx**
5. Enter kernel-mode
   a. Magic happens
6. notepad.exe running

So Many Strings

- Single-character strings (`char`)
- Multibyte-character strings
  - Rarely used, at least within C/C++ and WinAPI
- Wide-character strings (`wchar_t`)
- T-char strings (`TCHAR`) - depends on `#define UNICODE`
- `ANSI_STRING` structures
- `UNICODE_STRING` structures

Single-character Strings

```
const char* my_string = "hello";
```

- ANSI C-style strings
- Each character is a single byte
- Null-terminated, `'\0'`

Wide-character Strings

```
const wchar_t* my_wide_string = L"hello";
```

- Each character is two bytes
  - Unicode support, UCS-16/UCS-2
- Null-terminated, `'\0'`
  - Two-byte NULL
- Typedef’d as **`WCHAR`** on Windows

ANSI_STRING structure

```
typedef struct {
    UINT16  Length;
    UINT16  MaximumLength;
    CHAR    *Buffer;
} ANSI_STRING;
```

- Single-character strings for the Native API and NT Kernel
- More-or-less only used for DOS device paths
- Handled using Rtl Ansi string functions

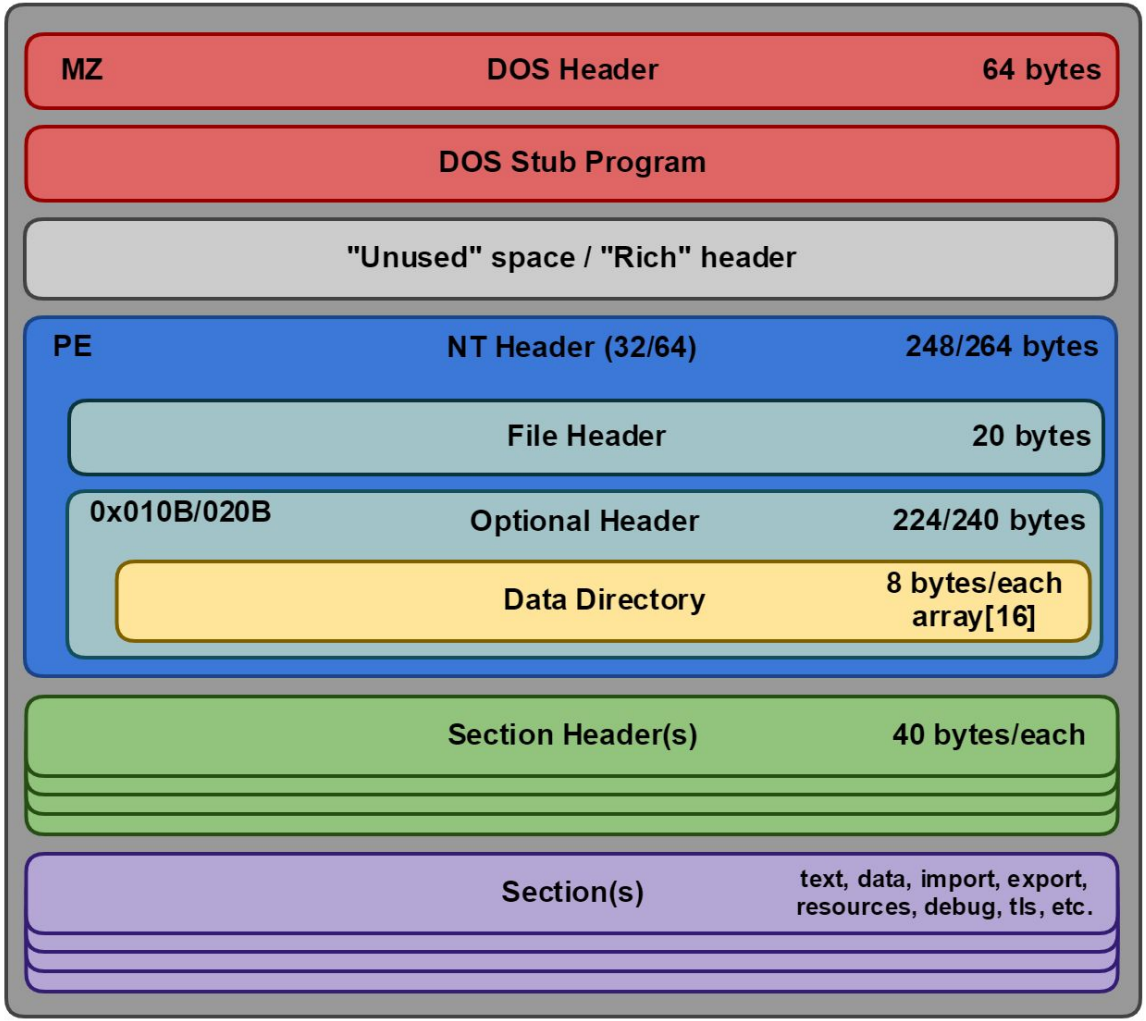
UNICODE_STRING structure

```
typedef struct {
    UINT16  Length;
    UINT16  MaximumLength;
    WCHAR   *Buffer;
} UNICODE_STRING;
```

- Most strings in the Native API and NT Kernel
- Managed with Rtl string functions: `RtlInitUnicodeString`, `RtlAppendUnicodeToString`, `RtlCompareUnicodeString`

Review: PE Format
&
Packed PEs

PE Format

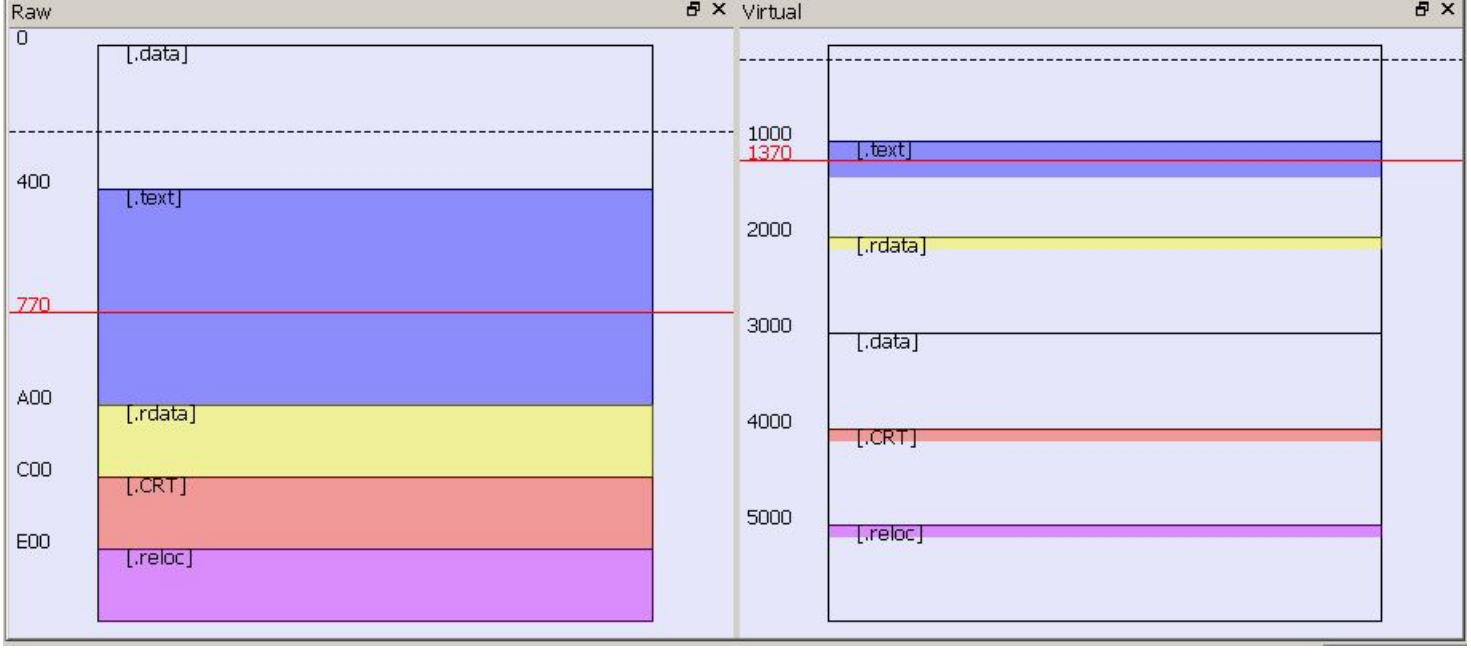

MZ
DOS Header
64 bytes
DOS Stub Program
"Unused" space / "Rich" header
PE
NT Header (32/64)
248/264 bytes
File Header
20 bytes
0x010B/020B
Optional Header
224/240 bytes
Data Directory
8 bytes/each
array[16]
Section Header(s)
40 bytes/each
Section(s)
text, data, import, export,
resources, debug, tls, etc.

Executing/Loading a PE

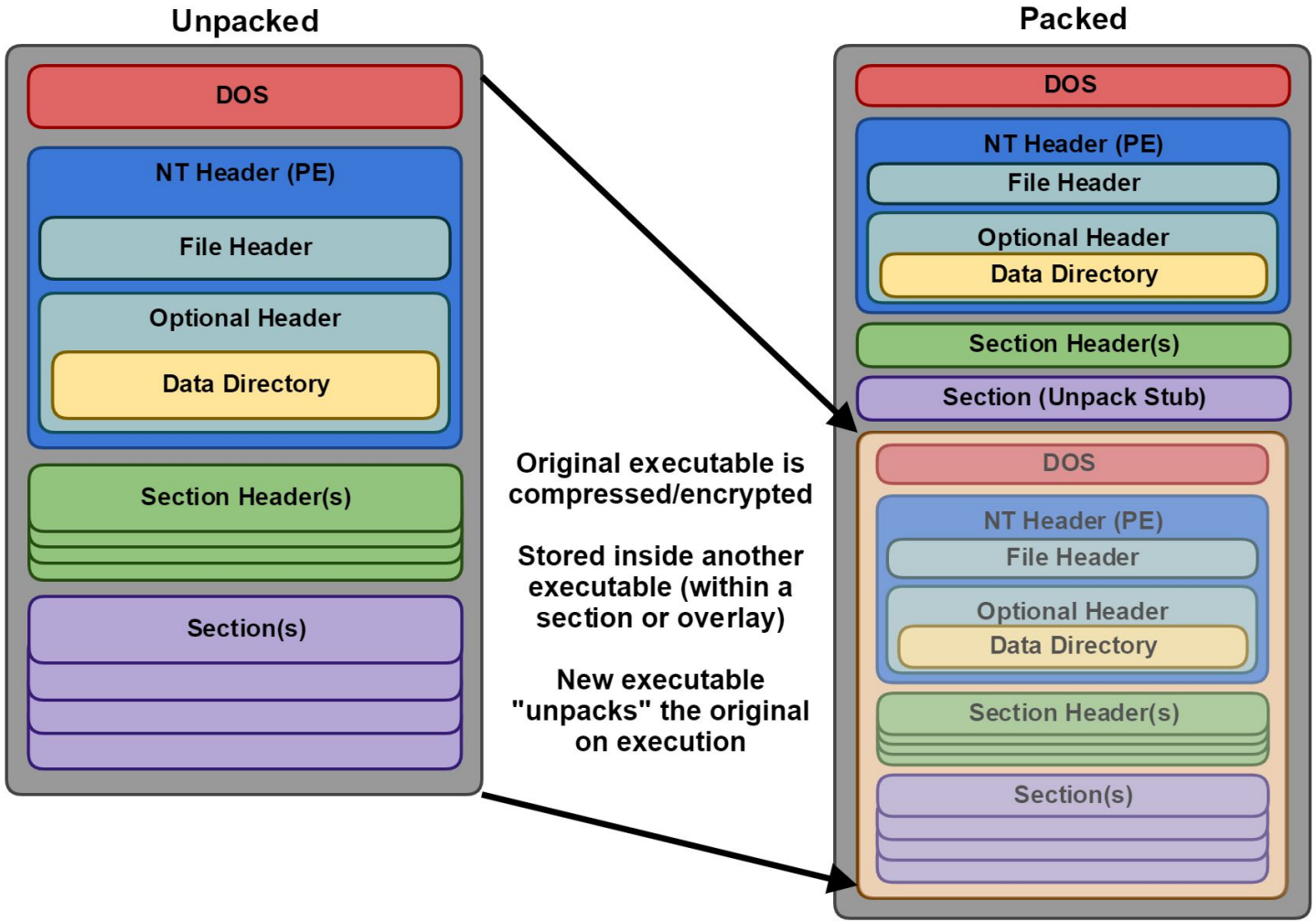
- Image Loader lives in *NTDLL.DLL* (Ldr functions)
  a. Loader maps PE headers at a determined **Image Base**
  b. PE headers are parsed
  c. Sections mapped into process address space
  d. Import directories and export tables used to resolve functions
     - If a new dependency is needed, load and proceed to (a)
- Execute PE entry point

Section Mapping Visualization (PE-bear)

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	600	1000	412	60000020	0	0	0
.rdata	A00	200	2000	2A	40000040	0	0	0
.data	0	0	3000	14	C0000040	0	0	0
.CRT	C00	200	4000	C	40000040	0	0	0
.reloc	E00	200	5000	40	42000040	0	0	0

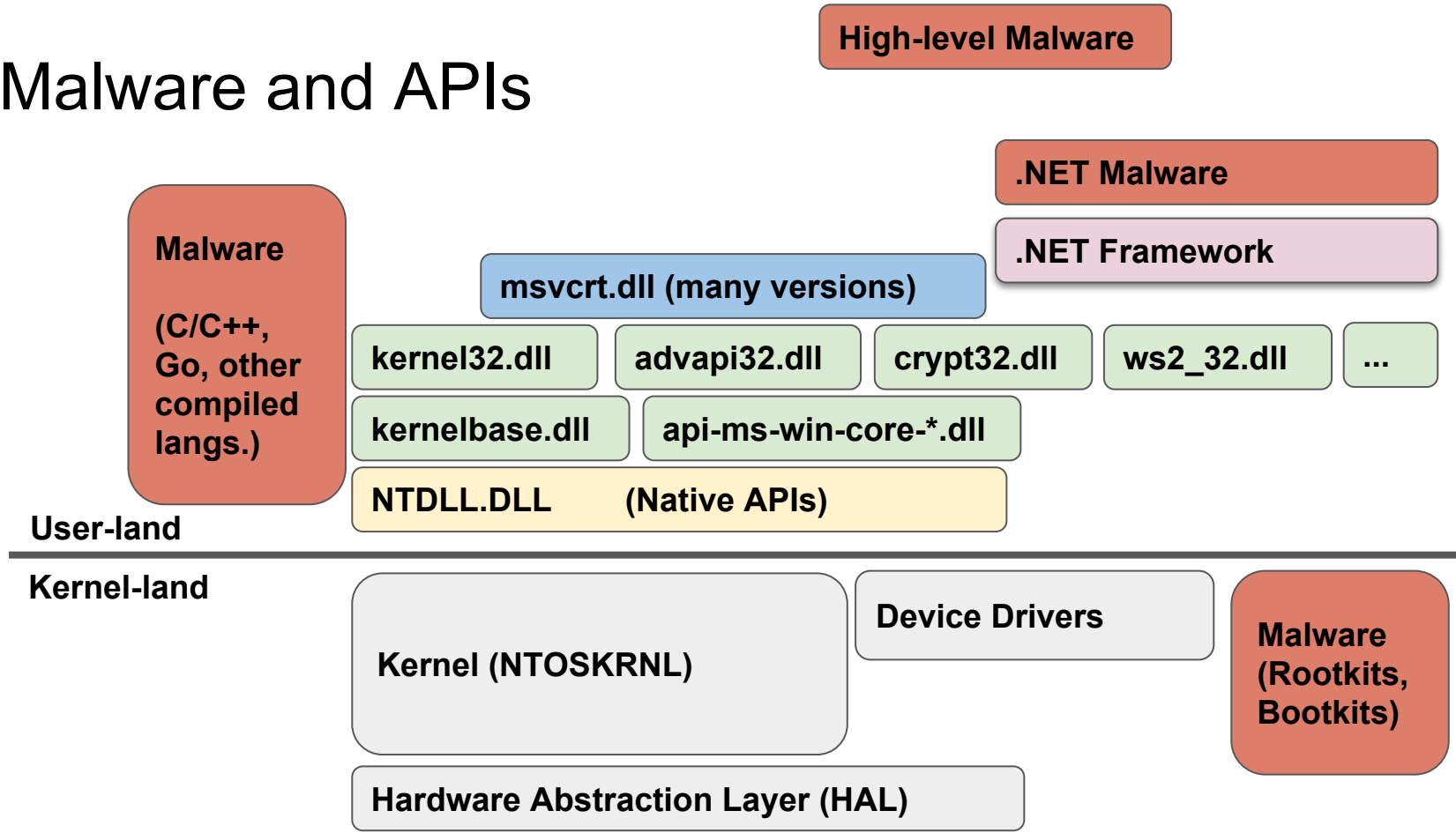


00400000	00001000	hello_stealth_original.exe		IMG	-R---	ERWC-
00401000	00001000	".text"	Executable code	IMG	ER---	ERWC-
00402000	00001000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00403000	00001000	".data"	Initialized data	IMG	-RW--	ERWC-
00404000	00001000	".tlsc"	Thread-local storage	IMG	-R---	ERWC-
00410000	000C5000	\Device\HarddiskVolume4\Windows\S		MAP	-R---	-R---
00590000	00007000			PRV	-RW--	-RW--
00597000	00009000	Reserved (00590000)		PRV		-RW--
00700000	00005000			PRV	-RW--	-RW--
00705000	000FB000	Reserved (00700000)		PRV		-RW--
766F0000	00001000	kernelbase.dll		IMG	-R---	ERWC-
766F1000	001C3000	".text"	Executable code	IMG	ER---	ERWC-
768B4000	00004000	".data"	Initialized data	IMG	-RW--	ERWC-
768B8000	00006000	".idata"	Import tables	IMG	-R---	ERWC-
768BE000	00001000	".didat"		IMG	-R---	ERWC-
768BF000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
768C0000	0002A000	".reloc"	Base relocations	IMG	-R---	ERWC-
77710000	00001000	kernel32.dll		IMG	-R---	ERWC-
77711000	0000F000	Reserved (77710000)		IMG		ERWC-
77720000	00064000	".text"	Executable code	IMG	ER---	ERWC-
77784000	0000C000	Reserved (77710000)		IMG		ERWC-
77790000	0002F000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
777BF000	00001000	Reserved (77710000)		IMG		ERWC-
777C0000	00001000	".data"	Initialized data	IMG	-RW--	ERWC-
777C1000	0000F000	Reserved (77710000)		IMG		ERWC-
777D0000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
777D1000	0000F000	Reserved (77710000)		IMG		ERWC-
777E0000	00005000	".reloc"	Base relocations	IMG	-R---	ERWC-
777E5000	0000B000	Reserved (77710000)		IMG		ERWC-
77920000	00009000			IMG	-R---	ERWC-
77930000	00001000	ntdll.dll		IMG	-R---	ERWC-
77931000	0011C000	".text"	Executable code	IMG	ER---	ERWC-
77A4D000	00001000	"RT"		IMG	ER---	ERWC-
77A4E000	00006000	".data"	Initialized data	IMG	-RW--	ERWC-
77A54000	00003000	".mrdata"		IMG	-R---	ERWC-
77A57000	00001000	".00cfg"		IMG	-R---	ERWC-
77A58000	0006F000	".rsrc"	Resources	IMG	-R---	ERWC-
77AC7000	00005000	".reloc"	Base relocations	IMG	-R---	ERWC-
7FE50000	00005000			MAP	-R---	-R---


Unpacked
DOS
NT Header (PE)
File Header
Optional Header
Data Directory
Section Header(s)
Section(s)
Original executable is compressed/encrypted
Stored inside another executable (within a section or overlay)
New executable "unpacks" the original on execution
Packed
DOS
NT Header (PE)
File Header
Optional Header
Data Directory
Section Header(s)
Section (Unpack Stub)
DOS
NT Header (PE)
File Header
Optional Header
Data Directory
Section Header(s)
Section(s)

Abusing the WinAPI

Malware and APIs



PE Unpacking and Executing

1. `memAddr = VirtualAlloc(codeBytes, READ|WRITE)`
2. `memcpy(memAddr, codeBytes, &codeBlob)`
   a. Often decrypted or decompressed at this stage
3. `VirtualProtect(memAddr, READ|EXECUTE)`
   a. Optional if allocated READ|WRITE|EXECUTE
4. `jmp memAddr`

Read/Writing Memory in Other Processes

- Virtual memory address-space is unique to a process
  - How do you access 0x004001000 in another process then?
- **ReadProcessMemory**
  - NtReadVirtualMemory
- **WriteProcessMemory**
  - NtWriteVirtualMemory

Injection with LoadLibrary and Threads

```
1. mod = GetModuleHandle("kernel32")
2. funcAddr = GetProcAddress(mod, "LoadLibraryA")
3. OpenProcess(THREAD_PERMS|VM_OP, procID)
4. memAddr = VirtualAllocEx(proc, READ|WRITE)
5. WriteProcessMemory(proc, memAddr, "C:\foo.dll")
6. CreateRemoteThread(proc, funcAddr, memAddr)
```

Cons: requires a DLL on disk, special DllMain, DLL mapped

Injection with Shellcode and Threads

1. `proc = OpenProcess(THREAD_PERMS|VM_OP, procID)`
2. `memAddr = VirtualAllocEx(proc, READ|WRITE)`
3. `WriteProcessMemory(proc, memAddr, &codeBlob)`
4. `VirtualProtectEx(proc, memAddr, READ|EXECUTE)`
   a. Optional if allocated `READ|WRITE|EXECUTE`
5. `CreateRemoteThread(proc, memAddr, NULL)`

Cons: new thread, may require position-independent code

Injection with Shellcode and APCs

1. `proc = OpenProcess(THREAD_PERMS|VM_OP, procID)`
2. `memAddr = VirtualAllocEx(proc, READ|WRITE)`
3. `WriteProcessMemory(proc, memAddr, &codeBlob)`
4. `VirtualProtectEx(proc, memAddr, READ|EXECUTE)`
5. `thread = OpenThread(..., threadID)`
6. `QueueUserAPC(memAddr, thread, NULL)`

Cons: may require position-independent code, tricky execution

Process Hollowing - “RunPE”

1. `CreateProcessA("explorer.exe", SUSPENDED, &proc)`
2. `NtUnmapViewOfSection(proc, all sections)`
3. `memAddr = VirtualAllocEx(proc, READ|WRITE)`
4. `WriteProcessMemory(proc, memAddr, &codeBlob)`
5. `VirtualProtectEx(proc, memAddr, READ|EXECUTE)`
6. `SetThreadContext(thread, CONTEXT -> &memAddr)`
7. `ResumeThread(thread)`

WinAPI, Internal APIs, Native API - Which one?

- Malware uses all of above - often inconsistently
- Setting breakpoints on correct functions important
- Patterns for unpacking, injection, and hollowing
  - Virtual memory operations
  - Writing to memory
  - Start execution (APCs, threads, UI callbacks, etc.)

Demo: Process Hollowing
And
Dumping/Unpacking

RunPE Demo

- Demo borrowed from @hasherezade
- Hollows out *calc.exe*

crackme1_packed.exe

- IDA: Disassemble packed file
- x32dbg/Scylla: Unpack the executable
- IDA: Disassemble unpacked file
- Solve the crack-me

Fin.

Questions?

Slides, demos, and source code are available at:

https://github.com/apodlosky/reFundamentals/