Unpacking Malware, Trojans and Worms

PE Packers Used in Malicious Software

Presented by Paul Craig
Ruxcon 2006

security-assessment.com

Overview

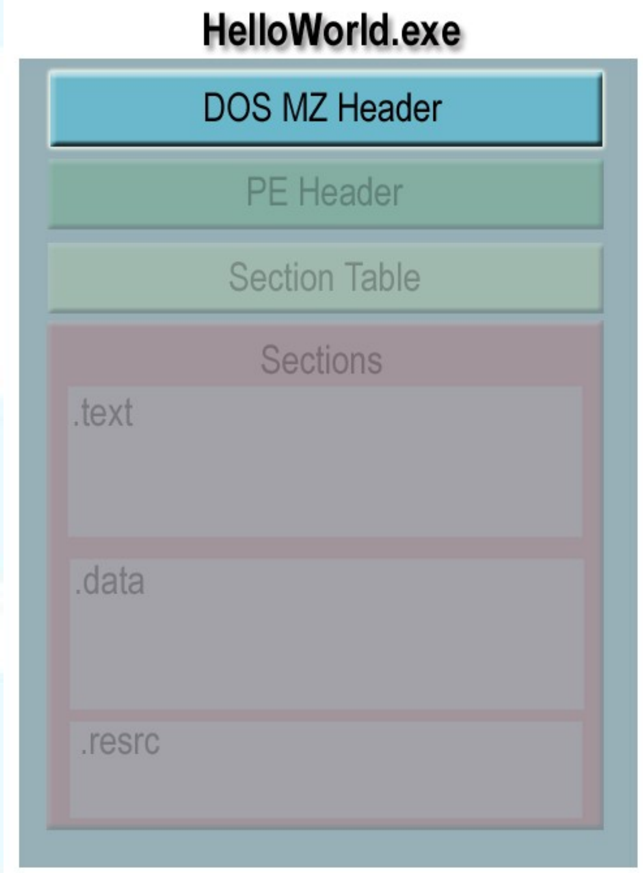
- Mental Refresher Course!
  - #1 - PE-COFF: The Windows Executable Format.
  - #2 - Who How What of Windows Import Address Tables.
- What is a PE Packer?
- PE Packers in Malicious Software.
- Detecting a PE Packer.
- Fundamental Weaknesses.
- PE Unpacking
- Automation
- Getting Tricky
- Unpacking NSPack
- Conclusion

Refresher #1 -

PE-COFF:
The Windows Executable Format.
Section-By-Section

PE COFF: Refreshing the mind.

- **DOS MZ Header**

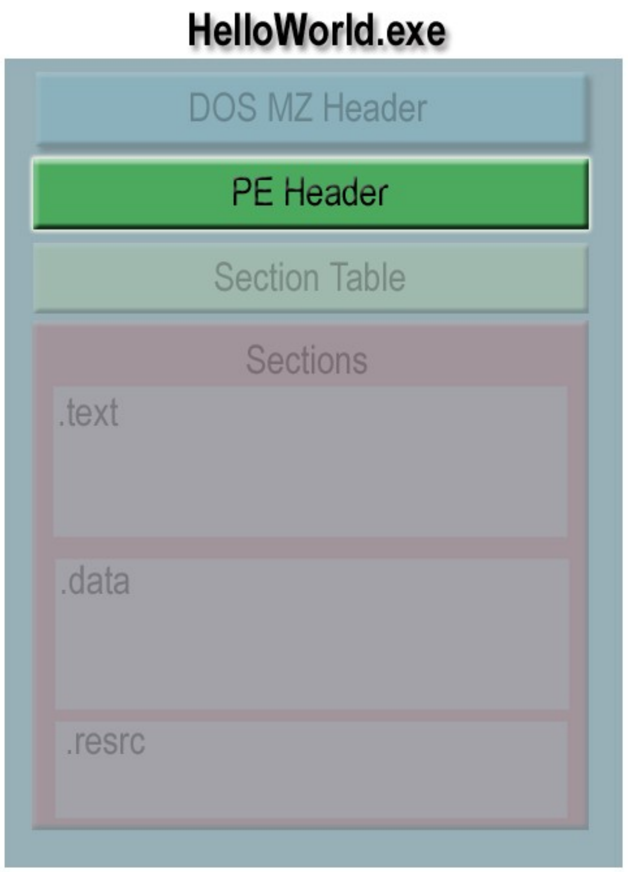


- Legacy Support for DOS
- 4D 5A (MZ) – Magic Number
  - Mark Zbikowski
- Check if being ran in DOS.
  - "This program must be ran under Windows."

- Handy to know that this string, is (almost always) in the DOS MZ header.

PE COFF: Refreshing the mind.

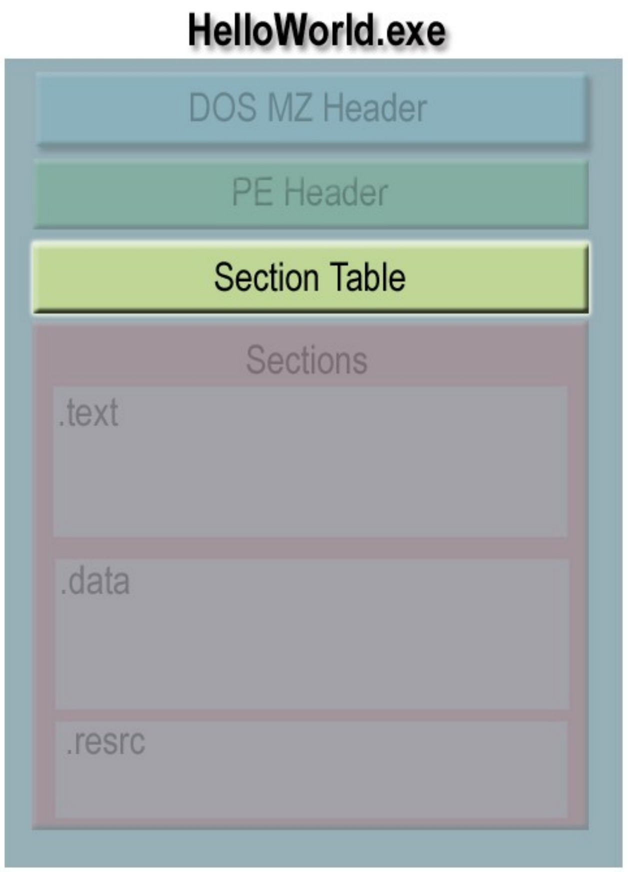
- **PE Header**



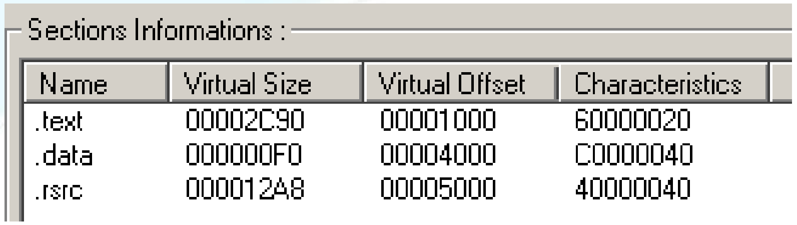
- 50h 45h 00h 00h – “PE”
- Data structures of execution settings
  - Machine Type
  - Date/Time Stamp
  - Size of executable
  - Where the code begins.

PE COFF: Refreshing the mind.

- **Section Table**



- A table of sections that exist inside the application.
- Each table has a name, permission and a size.
- When windows allocates memory pages for each the sections, the pages are set with the corresponding section permissions.

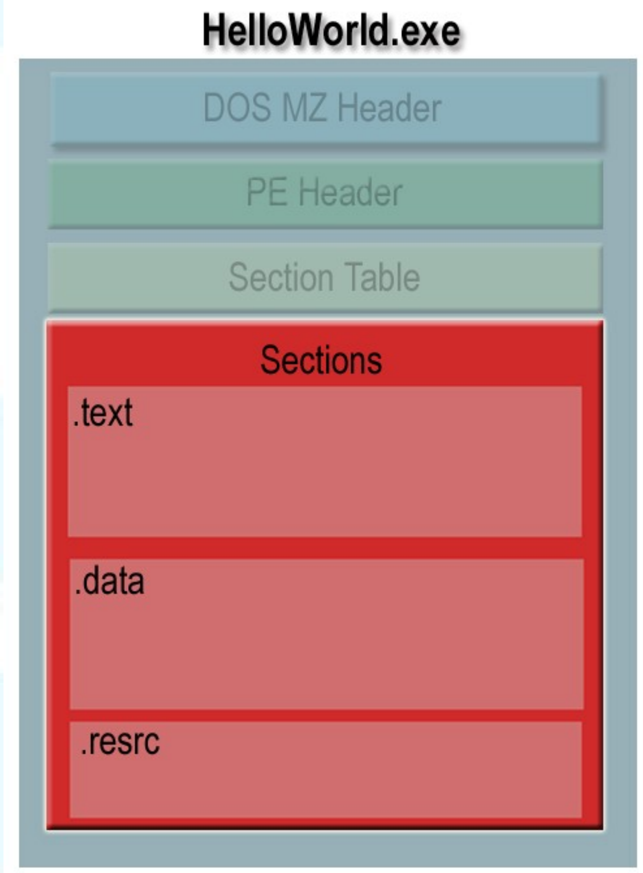


Sections Informations :

Name	Virtual Size	Virtual Offset	Characteristics
.text	00002C90	00001000	60000020
.data	000000F0	00004000	C0000040
.rsrc	000012A8	00005000	40000040

PE COFF: Refreshing the mind.

- **Sections**



- Executable sections are basically groups of different types of data.
- Grouping is based on common characteristics. IE is the data readable/writeable/executable?
- Compilers/linkers try to keep the number of sections as low as possible.
- Memory efficiency!

Refresher #2 -

The Who, How, What, Why of Windows Import Address Tables

Windows Import Address Table

- The Import Address Table is a table of external functions that an application wants to use.
  - Example, function Sleep(), from KERNEL32.DLL

- An Import Table will contain the location in memory of an imported function.
  - This is handy when we want to call the function.

- Applications use the Import Address Table to find other DLL’s in memory.

Windows Import Address Table

- One Problem though..
  - When the executable is compiled, and the Import Table is built, the compiler and linker do not know where in memory the particular DLL will be.
- The location is dynamic, dependant on operating system, service pack and any security patches that may be installed.
- We need Windows to tell us the location in memory at runtime. There is a good chance the location will be different location, PC to PC.

Windows Import Address Table

- When compiled, an executables Import Address Table contains NULL memory pointers to each function. It will have the name of the function, and what DLL it comes from, but that’s it.

- When we start to execute an application, Windows will find the Import Address Table location (from PE header), and overwrite our NULLS with the correct memory location for each function.

- Windows populates the Import Address Table for us, telling us where we can find each function.

- When we want to call an external function, we call a pointer to the value in the Import Address table.
- It becomes our lookup table.

Windows Import Address Table

- Example: Application wants to call GetProcAddress from KERNEL32.DLL.

```
PUSH EBP
CALL DWORD PTR [0041302C]   - (Call whatever is stored at 0041302C)
```

Look at the executable in a hex editor, the Import Table contains NULL's.

0041302C = 00 00 00 00

However, if we look at the same location once the application is running from inside a debugger, we see.

0041302C = AB 0C 59 7C

- Windows populated the Import Table with the correct value.
  - 7C590CAB = Location of GetProcAddress

Ok, enough mental refreshing!!

What is a PE packer?

What is a PE-Packer?

- Think of it as an executable file, inside another executable file. Which can be inside another executable file.
  - Think Russian dolls (Matryoshka).

- The executable file is ‘packed’ inside another executable file!

- When executed, the ‘outer’ executable will unpack the contents of the ‘inner’ executable into memory and execute it.

- The inner most executable is the ‘real’ executable!

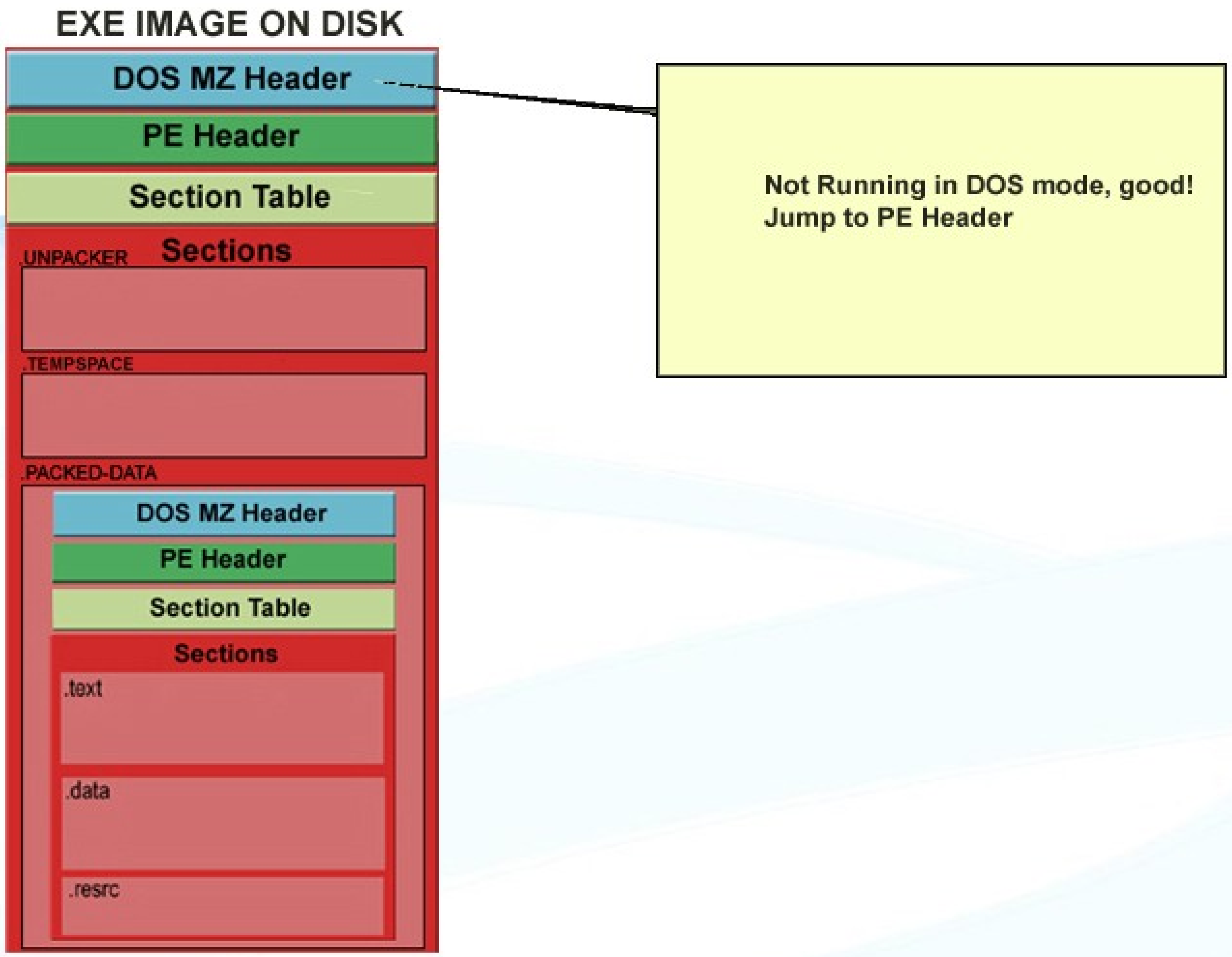
What is a PE-Packer?

- The first PE packers were designed as a method of reducing the size of an executable on disk, through compression.
  - Pklite from PKWARE (think PKZIP)
- PkLite ‘wraps’ around the target application, and compresses it.
- The packed executable is smaller on disk, but when ran will ‘unzip’ itself into memory.
- Once uncompressed in memory, the enclosed executable file is executed normally.

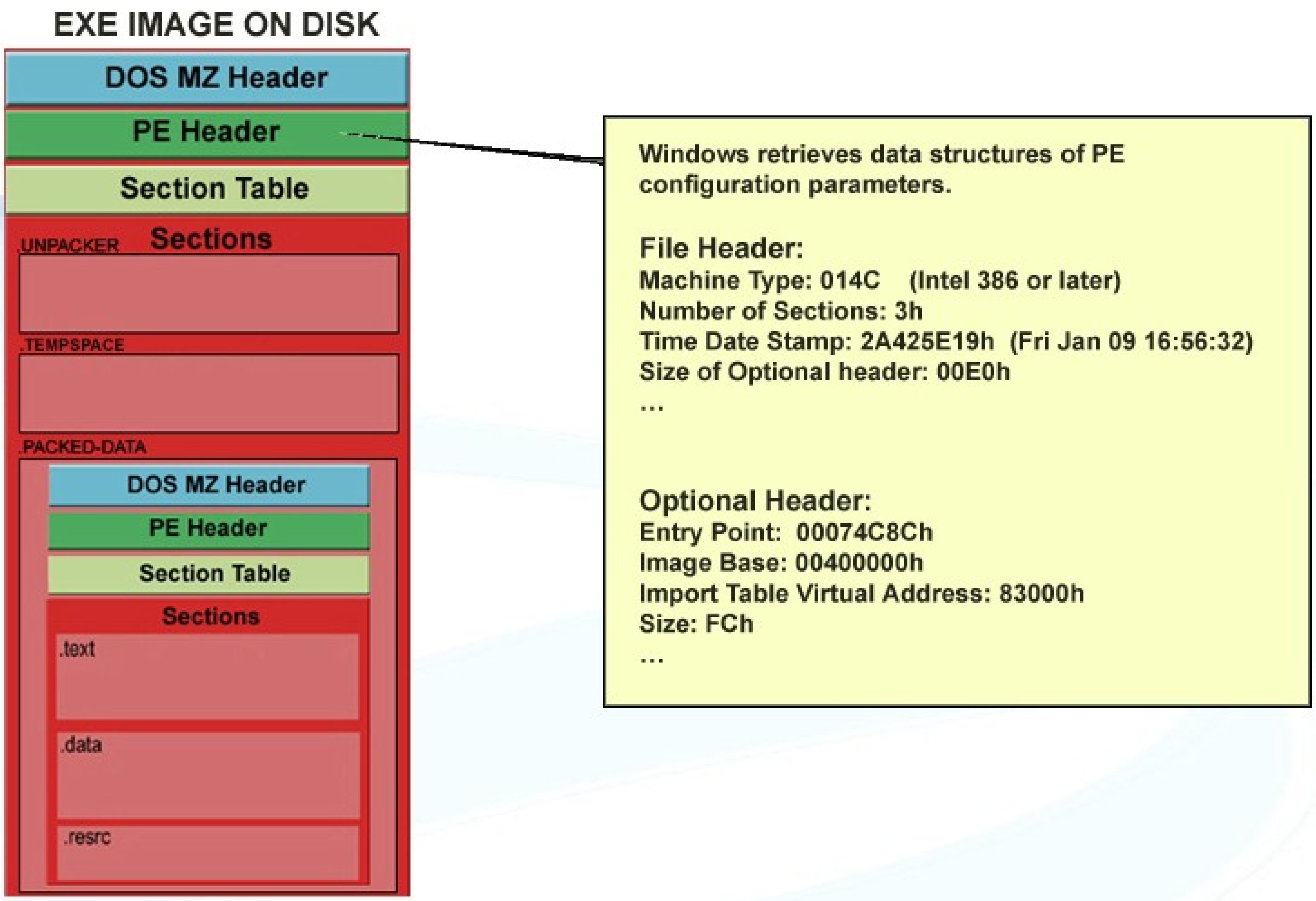
Step by Step PE Packer.

I just double clicked HelloWorld.exe
a packed .exe file.

What happens now?



DOS – MZ header

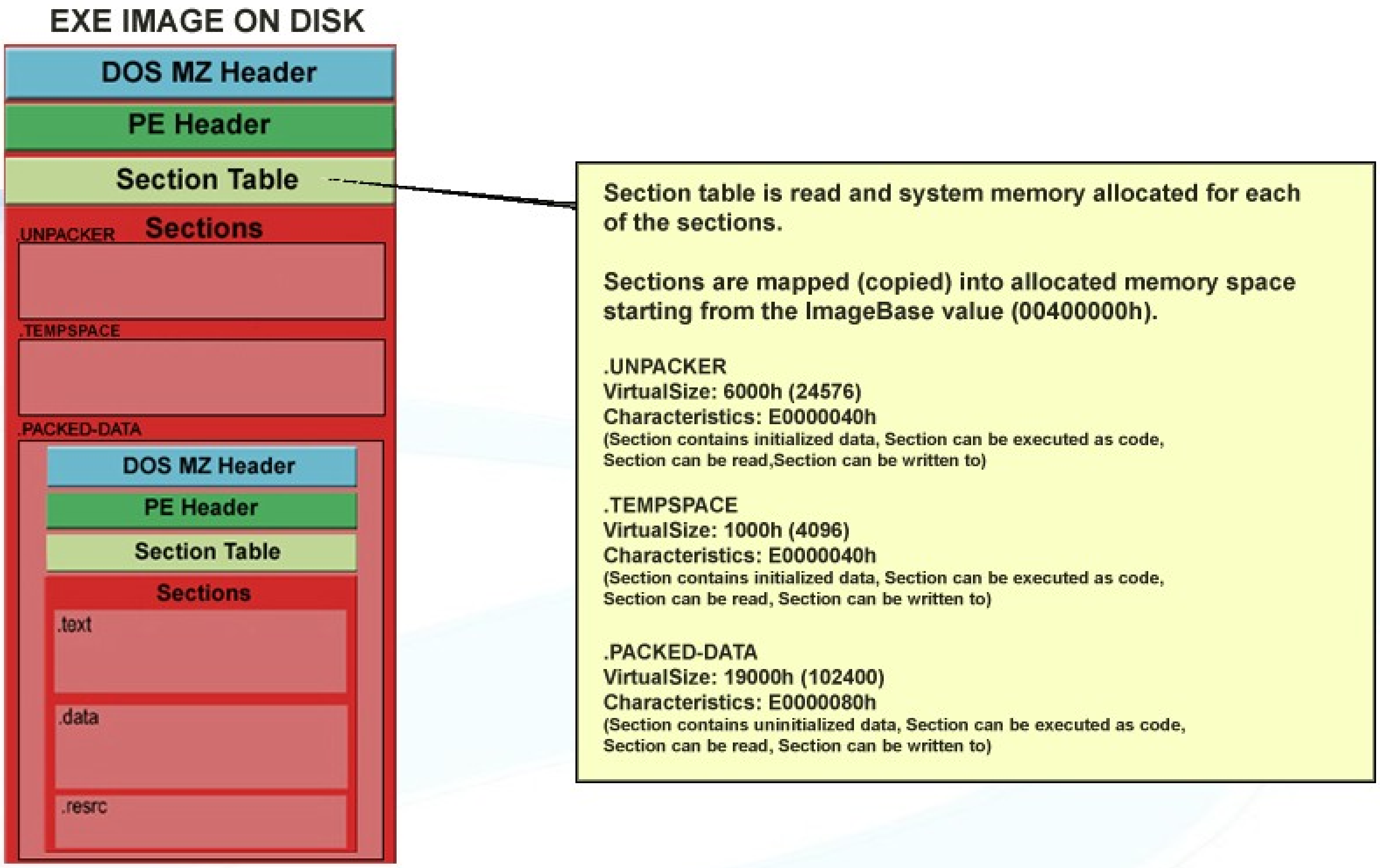


Windows retrieves data structures of PE configuration parameters.

File Header:
Machine Type: 014C (Intel 386 or later)
Number of Sections: 3h
Time Date Stamp: 2A425E19h (Fri Jan 09 16:56:32)
Size of Optional header: 00E0h
...

Optional Header:
Entry Point: 00074C8Ch
Image Base: 00400000h
Import Table Virtual Address: 83000h
Size: FCh
...

PE header



Section table is read and system memory allocated for each of the sections.

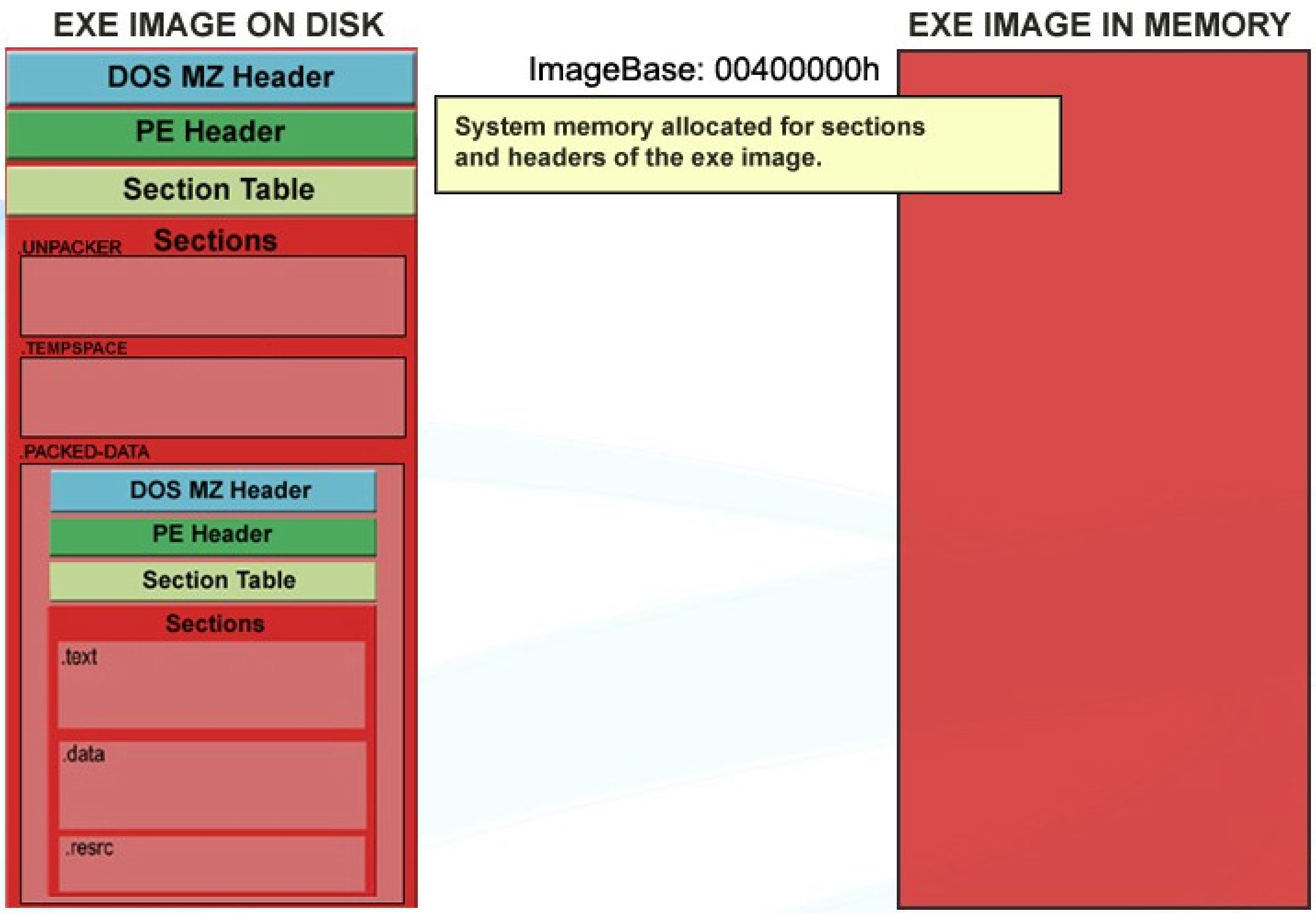
Sections are mapped (copied) into allocated memory space starting from the ImageBase value (00400000h).

.UNPACKER
VirtualSize: 6000h (24576)
Characteristics: E0000040h
(Section contains initialized data, Section can be executed as code, Section can be read,Section can be written to)

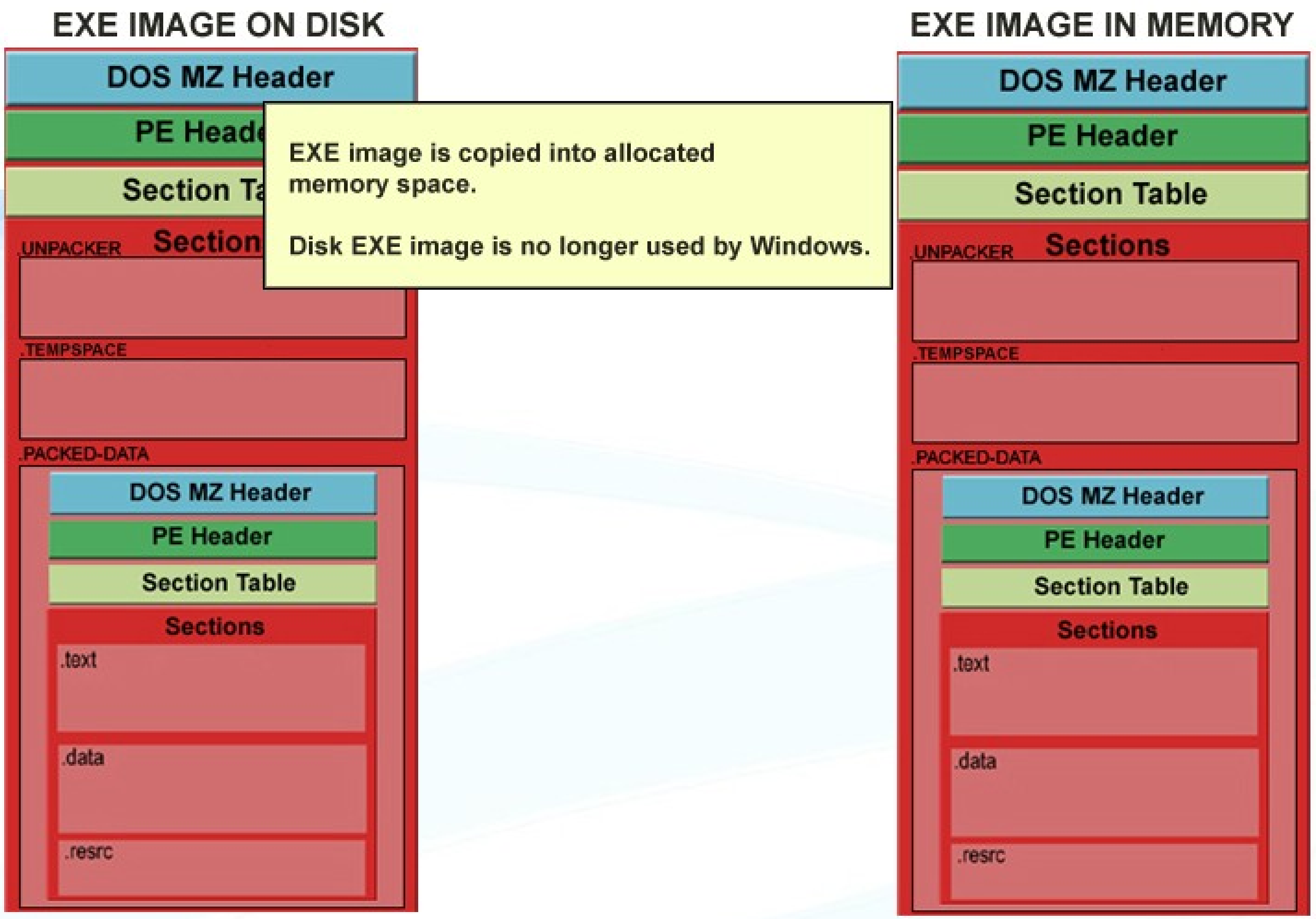
.TEMPSPACE
VirtualSize: 1000h (4096)
Characteristics: E0000040h
(Section contains initialized data, Section can be executed as code, Section can be read, Section can be written to)

.PACKED-DATA
VirtualSize: 19000h (102400)
Characteristics: E0000080h
(Section contains uninitialized data, Section can be executed as code, Section can be read, Section can be written to)

Windows reads section table



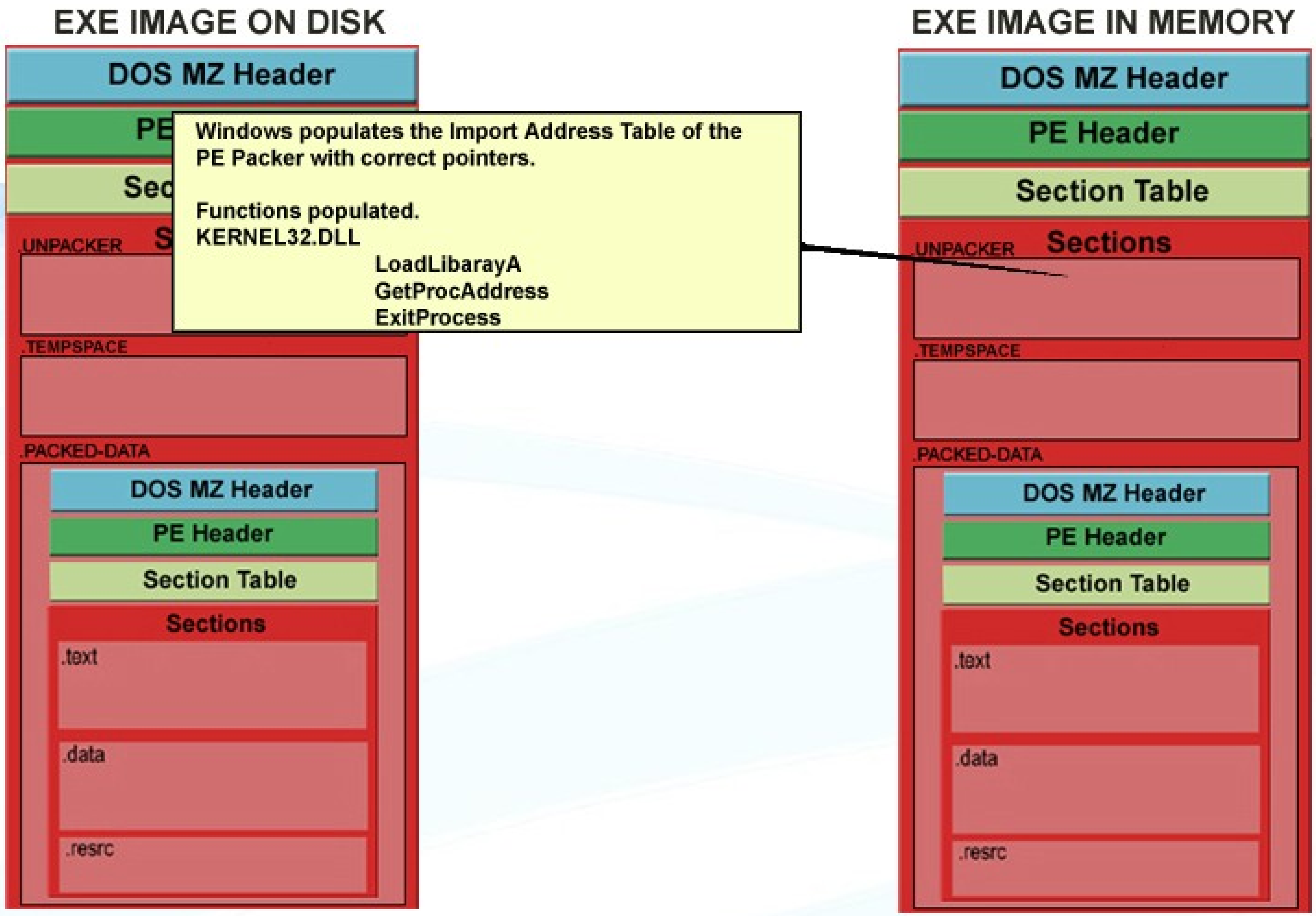
Memory allocated for executable



EXE image is copied into allocated memory space.

Disk EXE image is no longer used by Windows.

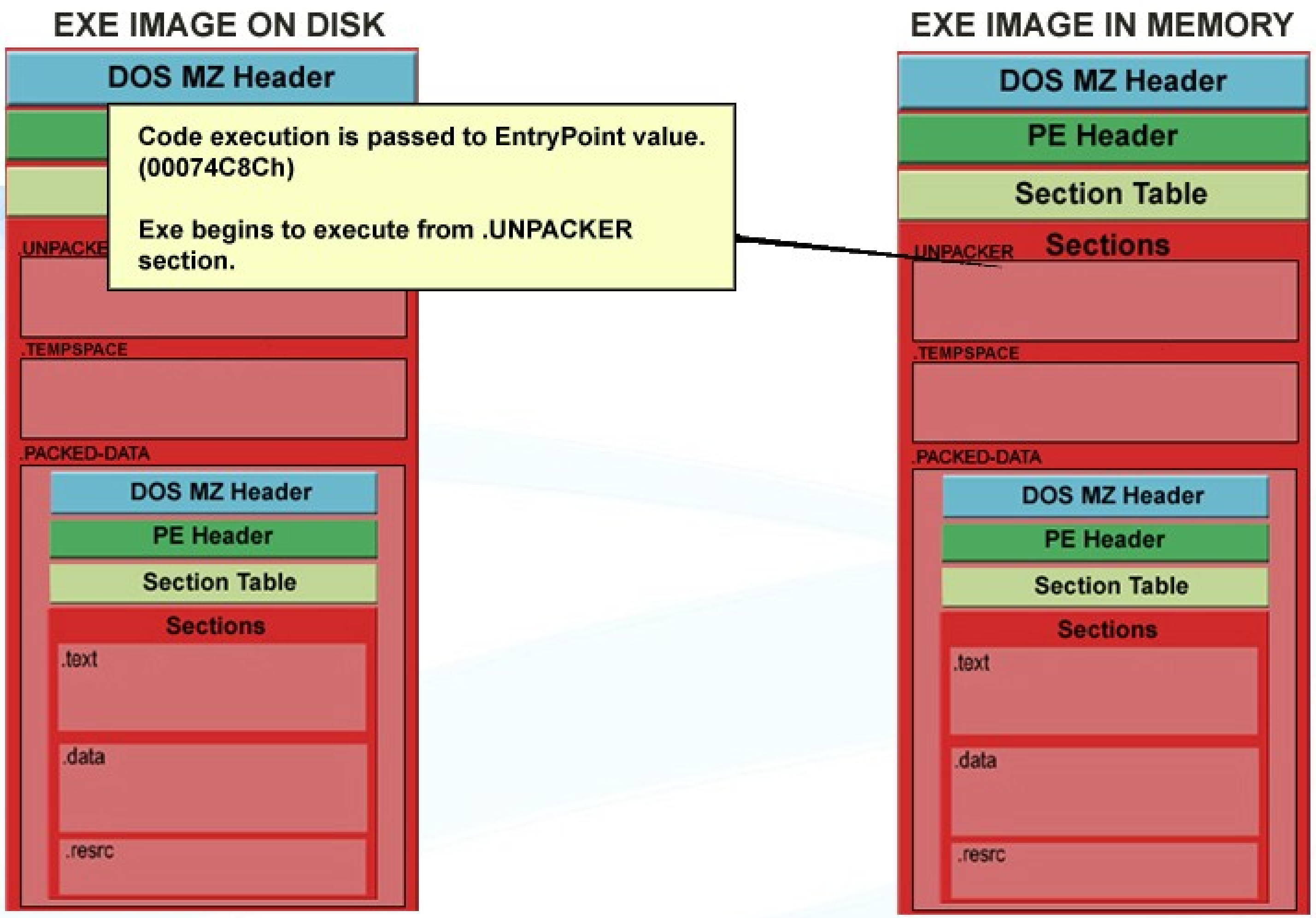
Disk image copied to memory



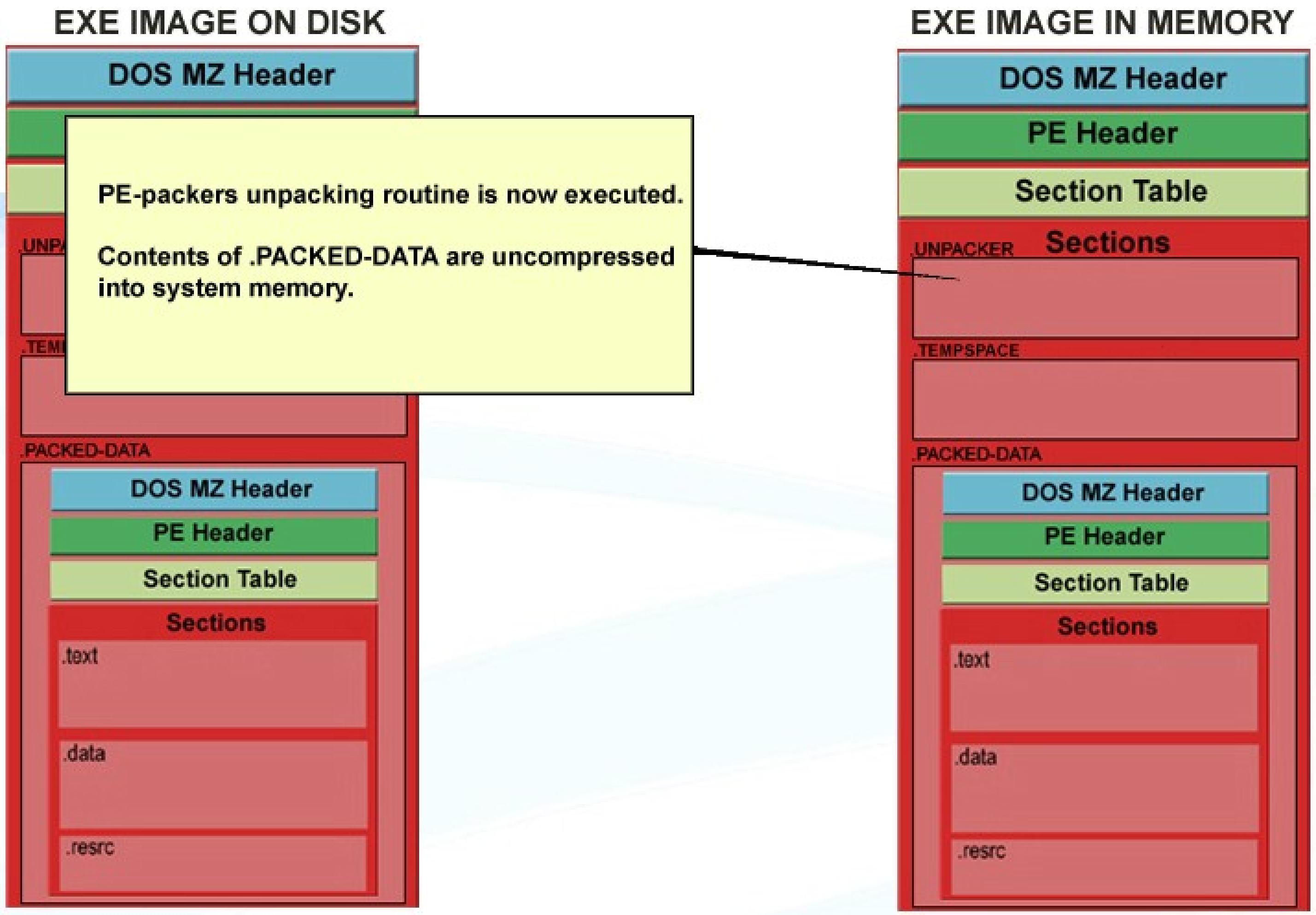
Windows populates the Import Address Table of the PE Packer with correct pointers.

Functions populated.
KERNEL32.DLL
LoadLibarayA
GetProcAddress
ExitProcess

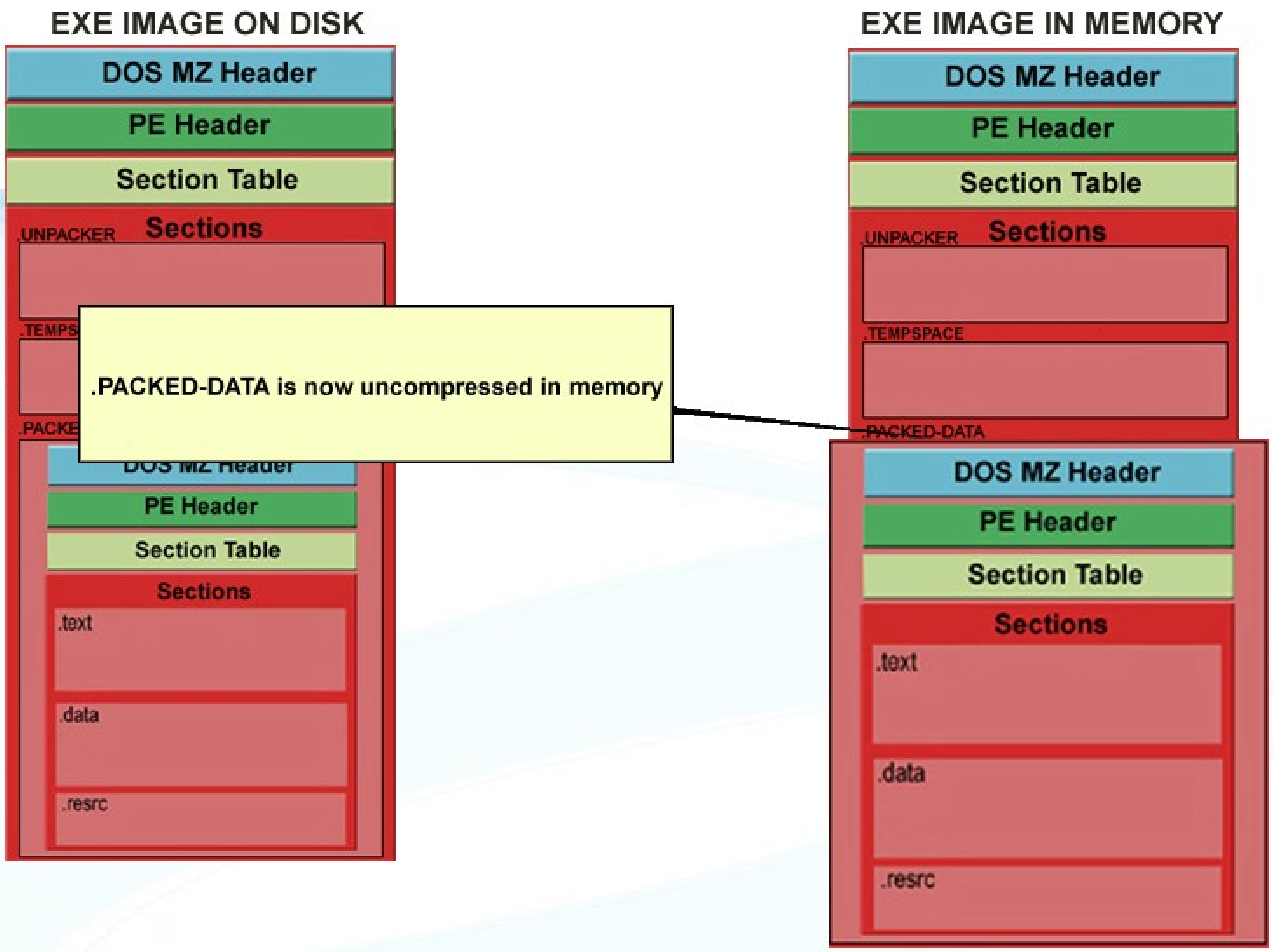
Windows populates IAT of PE packer



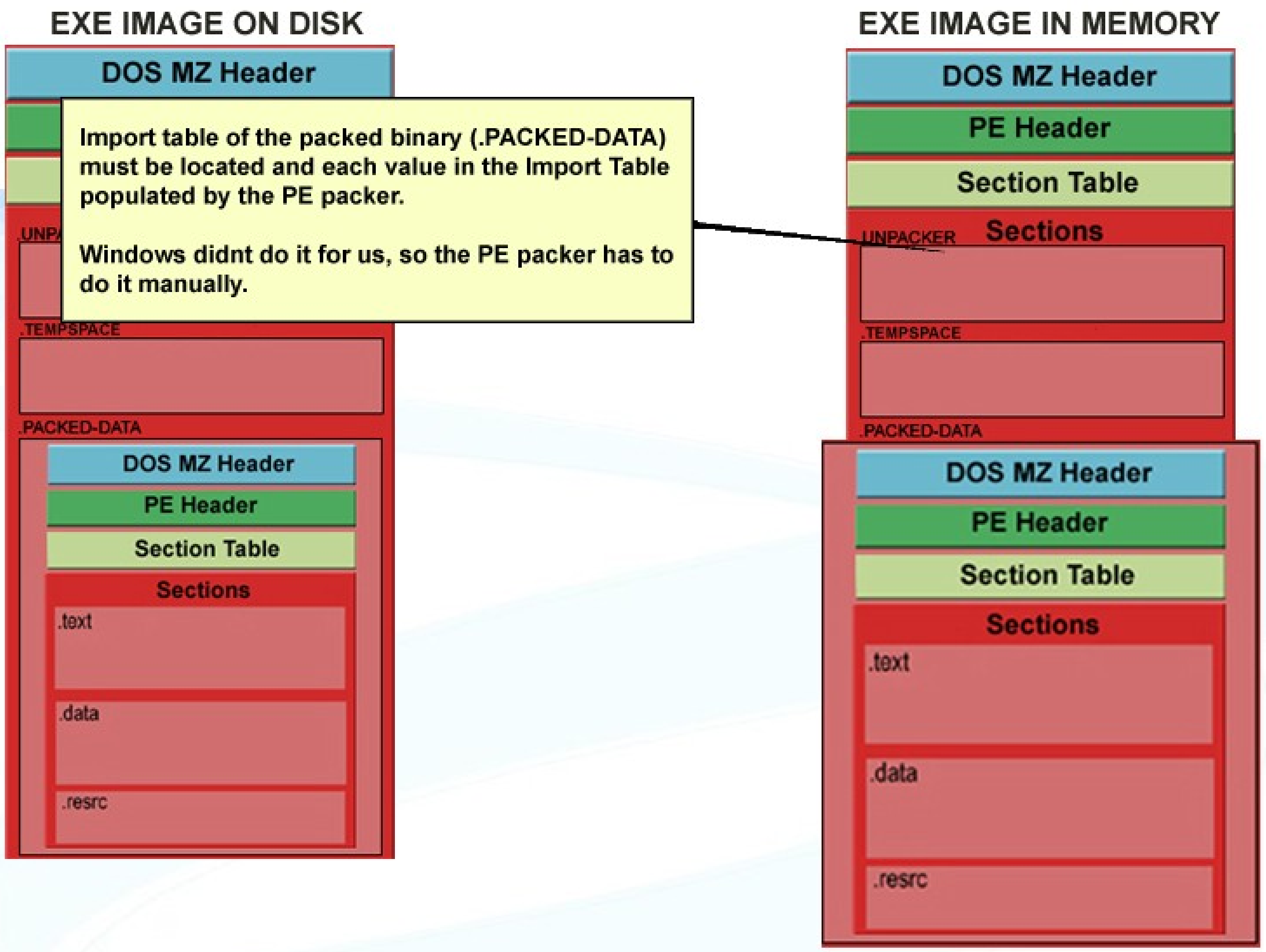
.UNPACKER section starts executing



.UNPACKER unpacks .PACKED-DATA into memory



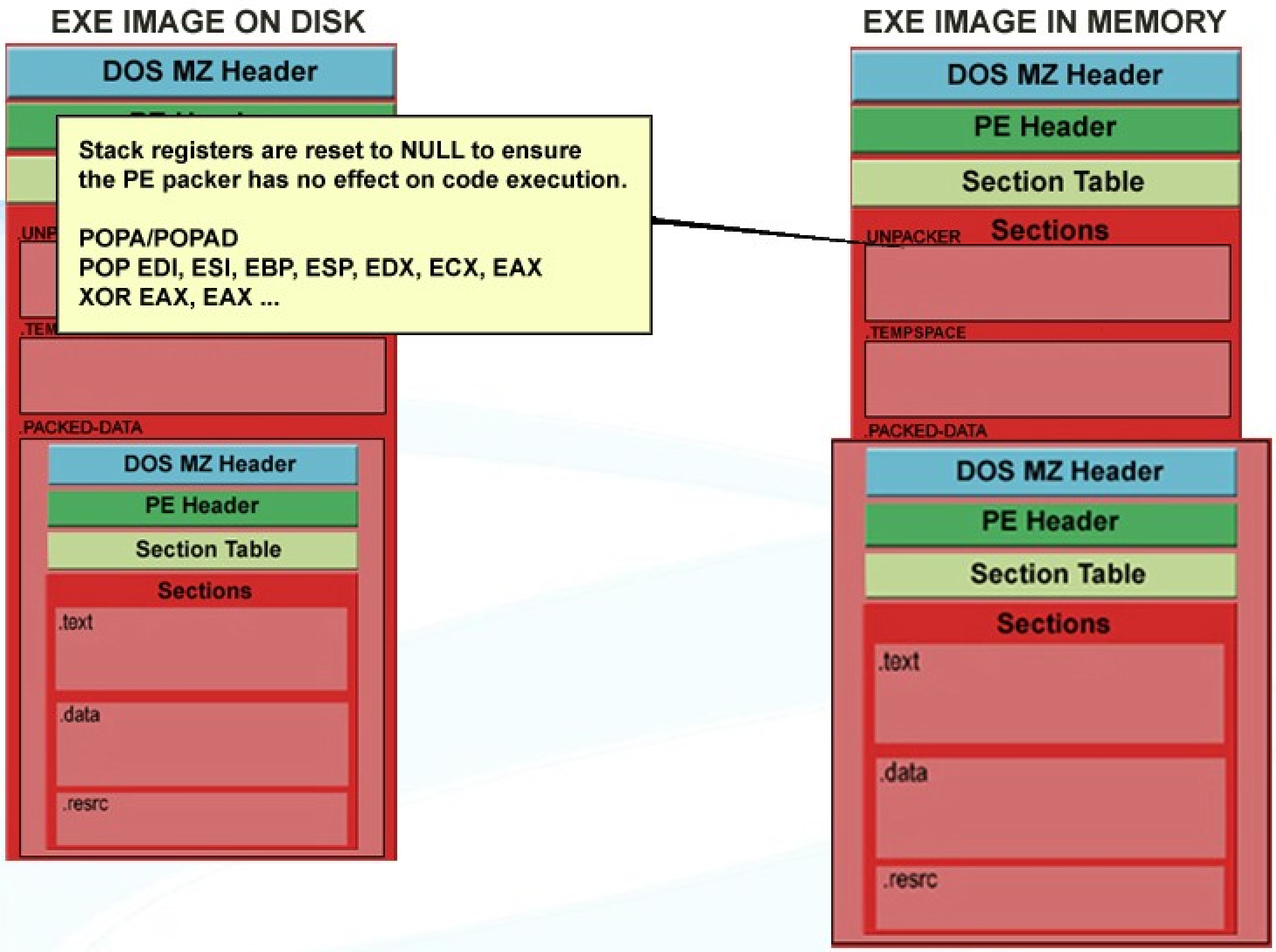
Unpacked, it is now larger in memory



Import table of the packed binary (.PACKED-DATA) must be located and each value in the Import Table populated by the PE packer.

Windows didnt do it for us, so the PE packer has to do it manually.

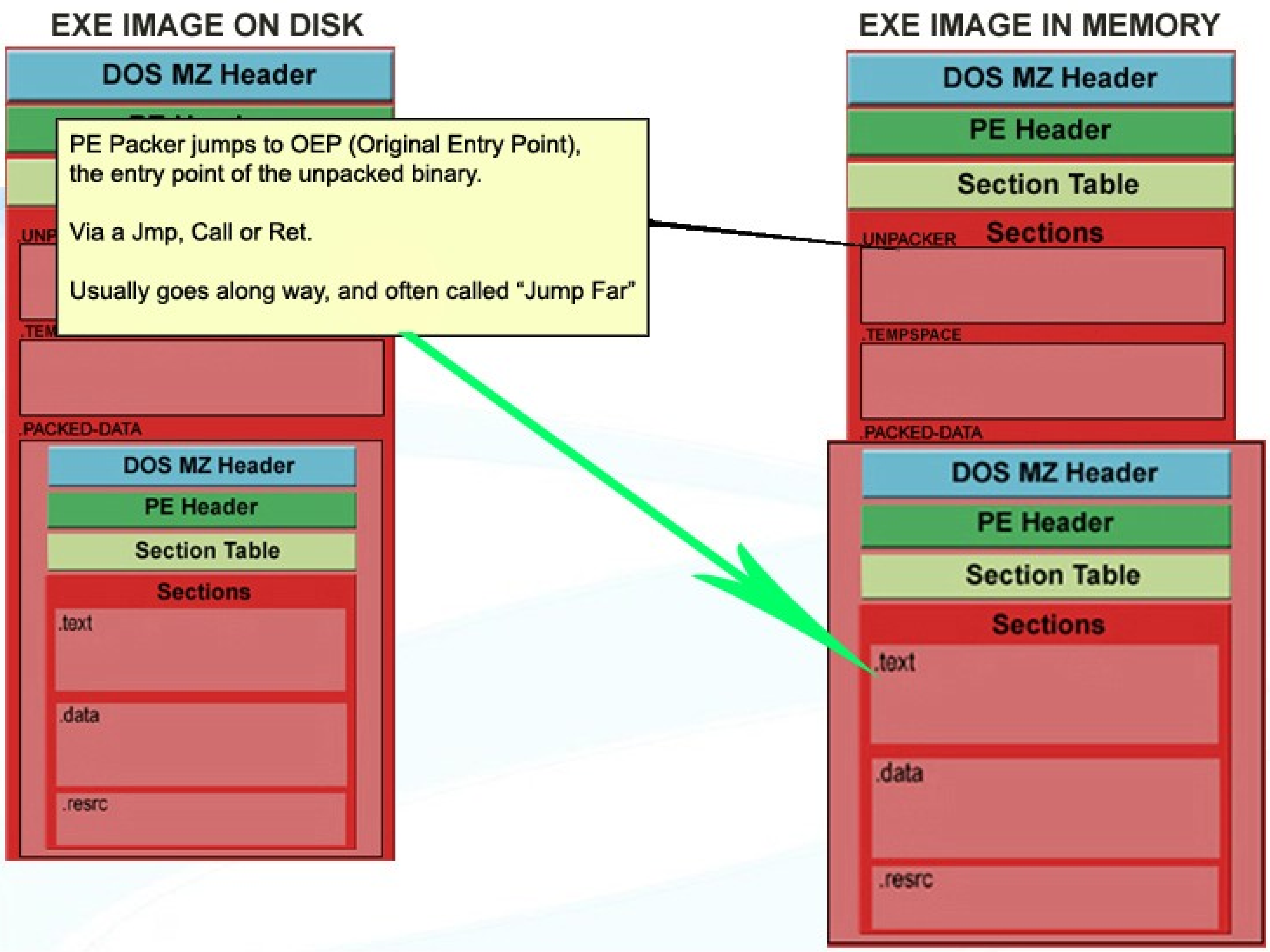
PE Packer populates Import Table



Stack registers are reset to NULL to ensure the PE packer has no effect on code execution.

POPA/POPAD
POP EDI, ESI, EBP, ESP, EDX, ECX, EAX
XOR EAX, EAX ...

Reset stack registers

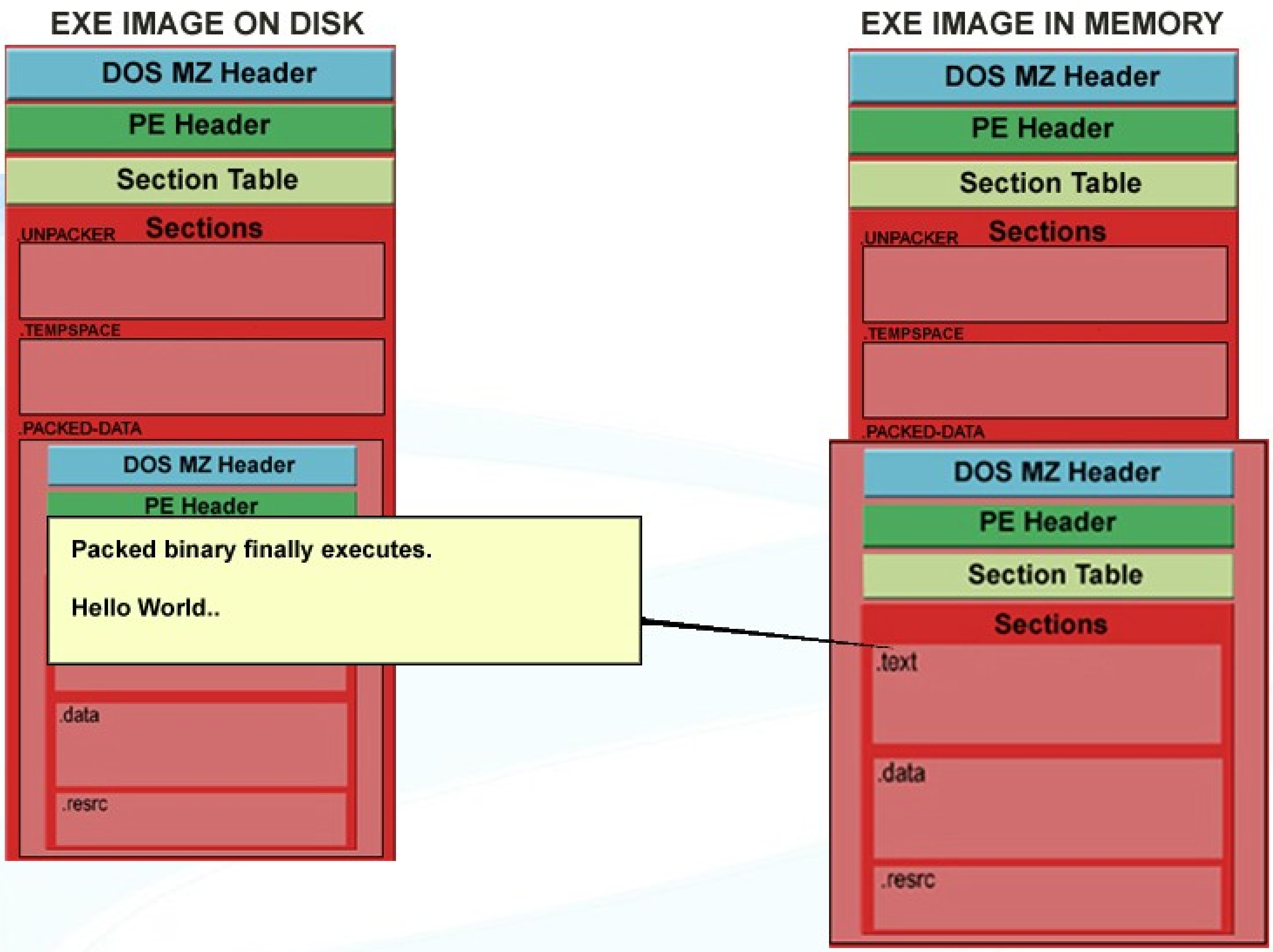


PE Packer jumps to OEP (Original Entry Point), the entry point of the unpacked binary.

Via a Jmp, Call or Ret.

Usually goes along way, and often called "Jump Far"

Jump to Original Entry Point (OEP)



Packed binary finally executes.

Hello World..

And it runs!

PE Packers In Malicious Software

- So what's the big deal with PE packing?

- Static analysis of PE packed data is not possible.
  - The payload is only unpacked at runtime!
  - It physically does not exist on disk, only in memory.

- Packed binaries can evade signature based AV.
  - A malicious executable is hiding inside an innocent executable.

  - Unless a virus scanner uses sandbox technology, it can be practically impossible to determine what's 'inside' the executable.

PE Packers In Malicious Software

- Lets put it a different way.
- What did the snake eat for lunch?

PE Packers In Malicious Software

- I have a phobia of snakes!
- Look at the snake all you want, it is not going to help!
- Static analysis is impossible.
- The snake might have eaten a weapon of mass destruction for all we know!
- Analysis is only possible when the snake is ‘unpacked’.
- Trojans/malicious software rely on ‘Snakes’ to hide themselves from anti virus software.

Detecting a PE-Packer

Detecting a PE Packer

- The first step in PE unpacking is to detect if a PE packer is being used.

- Do not rely solely on automated tools.
  - They can be defeated, evaded.
  - Custom PE packers can be used which are unknown to the tool.

- Analyzing the PE header and executable layout will tell us more than enough.

Detecting a PE Packer

- Acting suspiciously draws suspicion!
  - ‘Only dodgy people act dodgy.’

  - There are 4 simple steps that will tell us if an exe has been packed.

  - #1 - Very small import table.
    - A large application that only uses a few imports?
    - Example: LoadLibaryA, GetProcAddress …
    - These functions are used to locate other functions.
    - IE, when populating an import table manually.

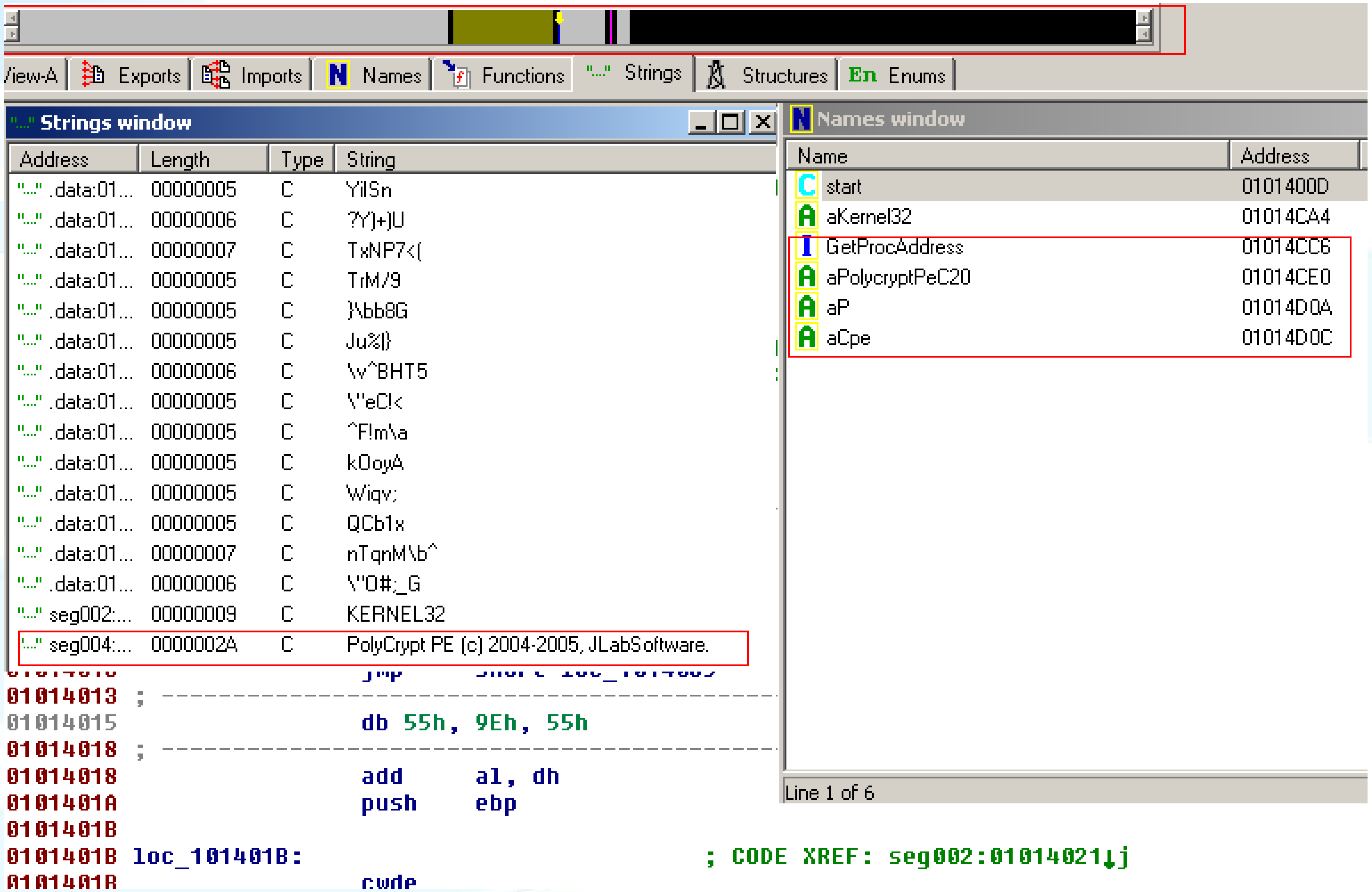
Detecting a PE Packer

- #2 - String table is missing or contains only garbage.
  - The string table is a table of commonly used strings in the application.
  - Strings stored in one location so the compiler/linker do not need to keep multiple copies of the string in memory.
  - A missing, corrupted or encrypted string table is usually a pointer that a PE packer has been used.
  - PE packers like to add entries into the string table.

Detecting a PE Packer

- #3 - Code body is far smaller than expected.
  - Remember, disassembly would only show the PE packer stub routine.
  - You will see large amounts of ‘data’ inside the executable.
  - Its packed, so we don’t see it as code

- #4 - Weird looking section names.
  - Compilers/linkers will try to have a standard naming convention for each code/data section.
  - Easy to spot something ‘strange’.



Address	Length	Type	String
.data:01...	00000005	C	YiISn
.data:01...	00000006	C	?Y)+)U
.data:01...	00000007	C	TxNP7<(
.data:01...	00000005	C	TrM/9
.data:01...	00000005	C	}\bb8G
.data:01...	00000005	C	Ju%l}
.data:01...	00000006	C	\v^BHT5
.data:01...	00000005	C	\"eC!<
.data:01...	00000005	C	^F!m\a
.data:01...	00000005	C	kOoyA
.data:01...	00000005	C	Wiqv;
.data:01...	00000005	C	QCb1x
.data:01...	00000007	C	nTqnM\b^
.data:01...	00000006	C	\"O#;_G
seg002:...	00000009	C	KERNEL32
seg004:...	0000002A	C	PolyCrypt PE (c) 2004-2005, JLabSoftware.

Name	Address
start	0101400D
aKernel32	01014CA4
GetProcAddress	01014CC6
aPolycryptPeC20	01014CE0
aP	01014D0A
aCpe	01014D0C

Line 1 of 6

```
01014013 ; ---------------------------------------------
01014015                 db 55h, 9Eh, 55h
01014018 ; ---------------------------------------------
01014018                 add     al, dh
0101401A                 push    ebp
0101401B
0101401B loc_101401B:                           ; CODE XREF: seg002:01014021↓j
0101401B                 cwde
```

Detecting a PE Packer

- Once we have analyzed the executable ourselves, THEN we then use PE scanning tools to help identify the packer that's being used.
  - PEiD (http://peid.has.it)
  - GT2 (http://www.programmerstools.com/)
- Tools can be wrong, don't be lazy.

The Fundamental Weakness.

“If it executes, we can unpack it.”

Fundamental Weakness

- No matter how an executable is packed, it MUST be unpacked at runtime for my CPU to run it!
  - My CPU has to run the plaintext, unpacked binary at some stage!
  - I don’t care if its packed using 2048bit RSA, my CPU only runs straight x86 ASM.

- Its all really about the timing.
  - If we want to get the unpacked data, we need to know the exact moment and location where the data will be unpacked and available.

- It may only be available and intact for a very short amount of time.

PE Unpacking

Objectives of PE Unpacking

- Re-create the executable, in its original form, before it was packed.
- This allows us to perform static analysis on the now unpacked ‘payload’ data.
- We should not need the PE packer stub again, so we can delete it.
- Bring the executable back to its virgin state.
  - Before it was packed.

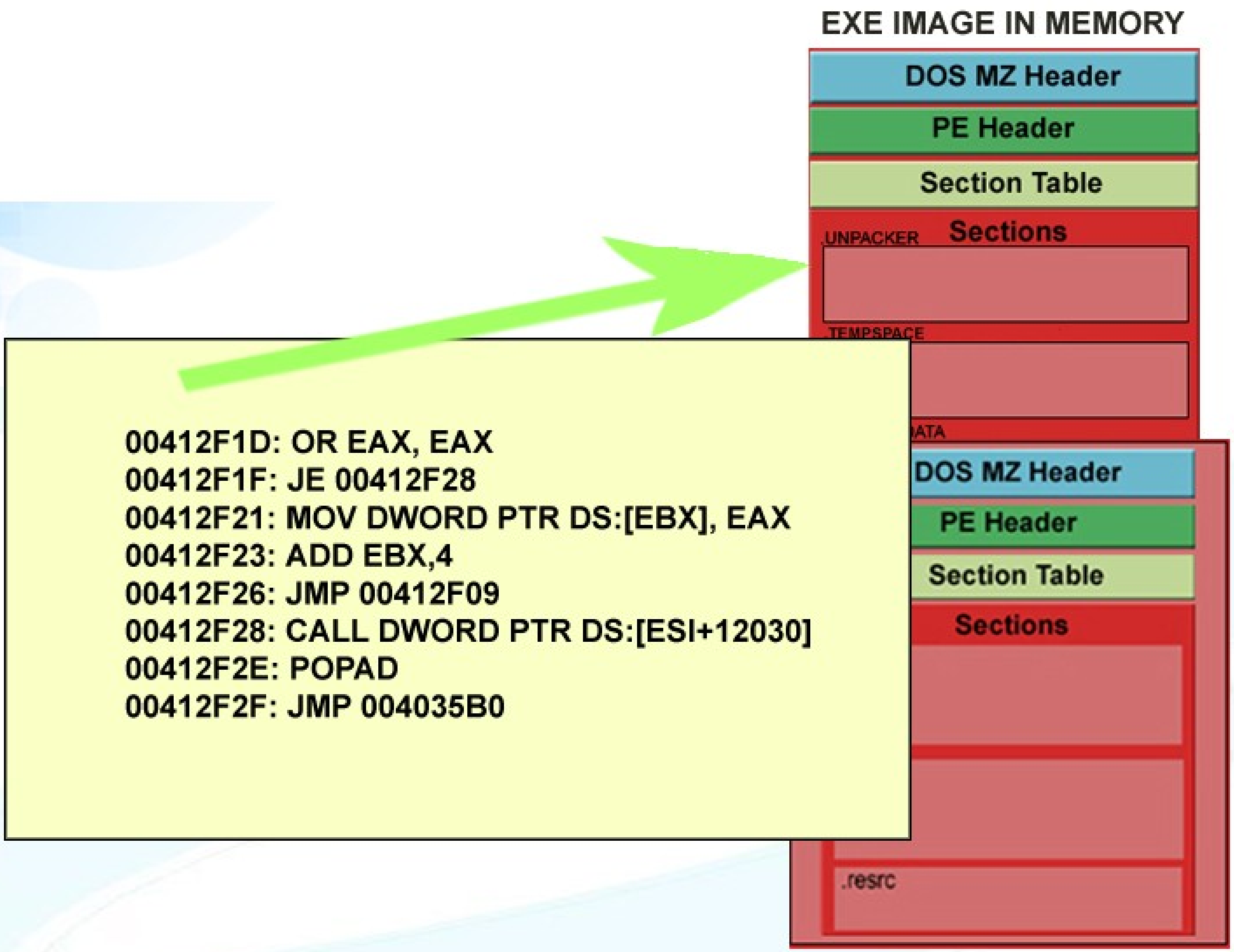
Objectives of PE Unpacking

Step #1.

Locate the OEP (Original Entry Point) jump.

- After the PE packer has finished unpacking itself and has populated the Import Address Table of the '.PACKED-DATA', it will usually reset/clear any stack registers it was using.
- Shortly after this, a jump/call will occur that will start the execution of the now unpacked data.
  - This is the OEP Jump
- The destination of this jump is the EntryPoint of the unpacked data!

It looks something like this.



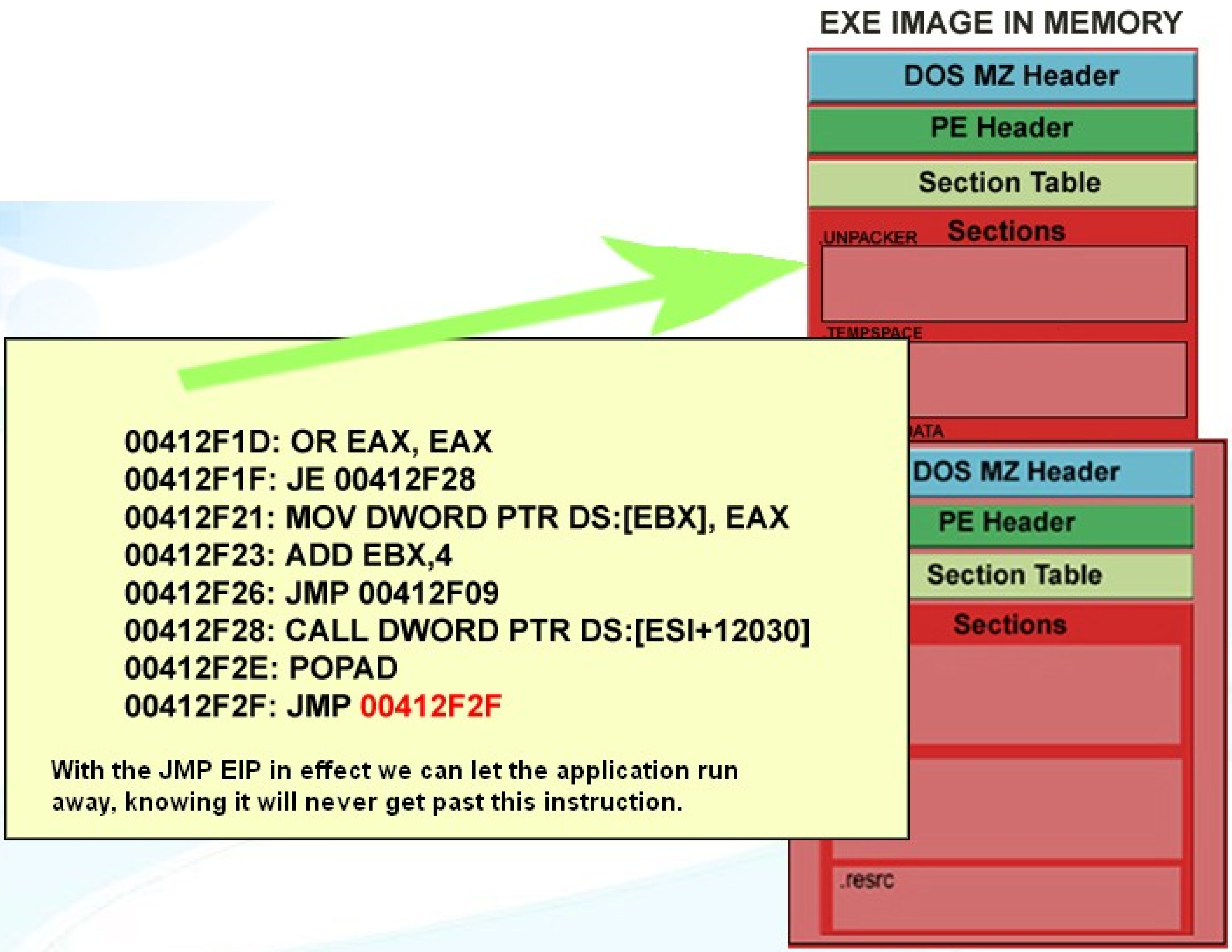
```
00412F1D: OR EAX, EAX
00412F1F: JE 00412F28
00412F21: MOV DWORD PTR DS:[EBX], EAX
00412F23: ADD EBX,4
00412F26: JMP 00412F09
00412F28: CALL DWORD PTR DS:[ESI+12030]
00412F2E: POPAD
00412F2F: JMP 004035B0
```

Objectives of PE Unpacking

Step #2 – For The Old School Only

- If your true old school you of-cause use Softice (A ring0 debugger). At this stage you need to manually suspend the application at the OEP JMP (Pause the process)
- Modify the OEP jump/call to be an infinite loop (JMP EIP).
- If you use OllyDBG, just ignore this.
- Guy on the left uses Softice, in-fact he use to work at Compuware.

-- Softice JMP EIP --



```
00412F1D: OR EAX, EAX
00412F1F: JE 00412F28
00412F21: MOV DWORD PTR DS:[EBX], EAX
00412F23: ADD EBX,4
00412F26: JMP 00412F09
00412F28: CALL DWORD PTR DS:[ESI+12030]
00412F2E: POPAD
00412F2F: JMP 00412F2F
```

With the JMP EIP in effect we can let the application run away, knowing it will never get past this instruction.

Objectives of PE Unpacking

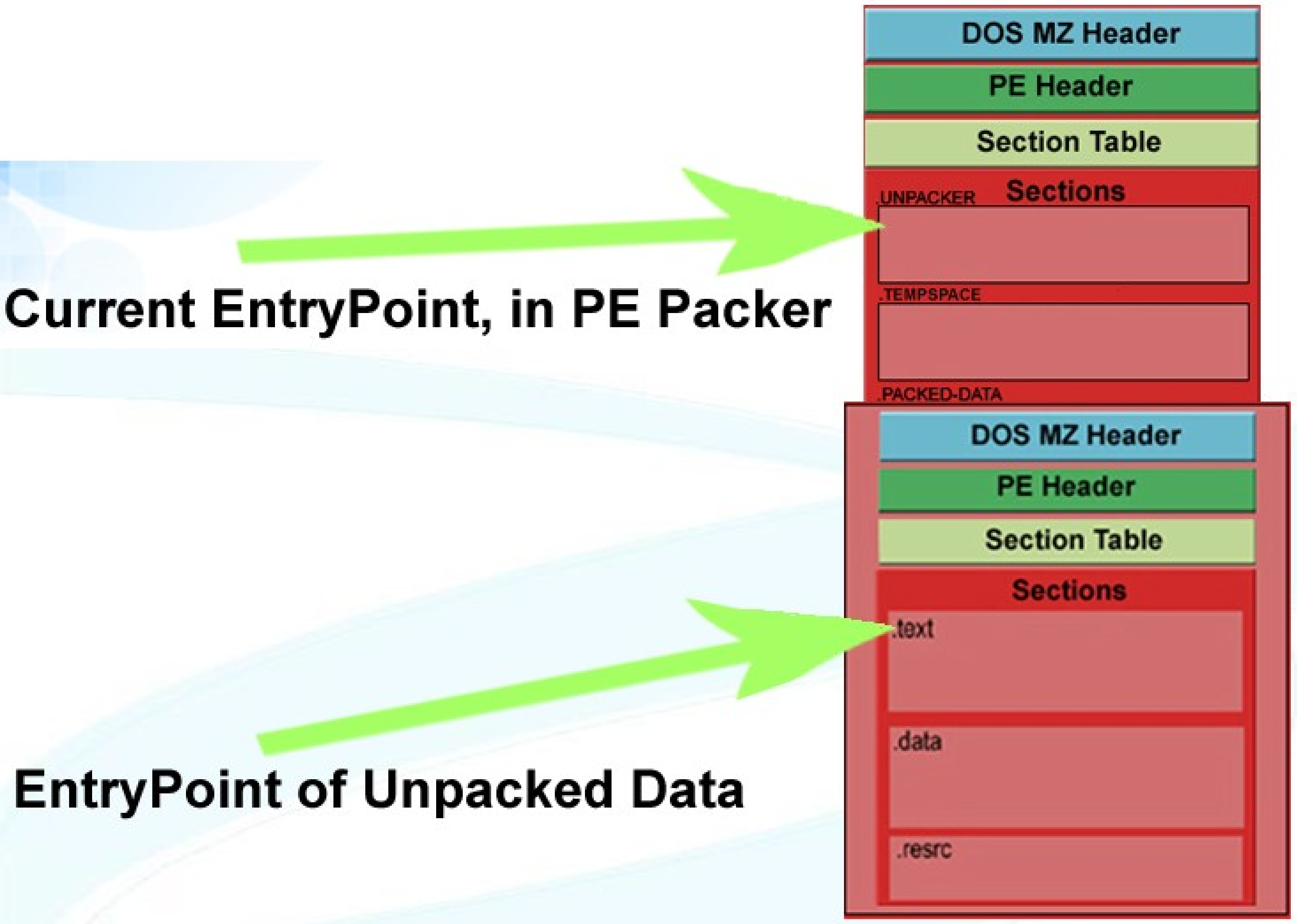
Step #3 :Dump the executable memory image

- The application is currently unpacked in memory, but has not yet begun to execute the unpacked data.
- We need to dump the memory image of the executable back to disk.
- We use a process dumping tool.
- After the memory image is dumped to disk we are left with a snapshot which contains both the unpacked (payload) data, and the PE packers 'unpacking' stub.

Objectives of PE Unpacking

Step #4 : Change EntryPoint of dumped image.

- The dumped executable’s EntryPoint still points to the start of the PE Packer, the ‘unpacking’ routine.
- We want the executable to start running the unpacked data first, not the PE packer, we don’t need the PE packer anymore!
- We know the Original EntryPoint is 004035B0h, this is where the PE packer was going to jump to.


DOS MZ Header
PE Header
Section Table
.UNPACKER
Sections
.TEMPSPACE
.PACKED-DATA
DOS MZ Header
PE Header
Section Table
Sections
.text
.data
.resrc
Current EntryPoint, in PE Packer
EntryPoint of Unpacked Data

Objectives of PE Unpacking

Step #4 Continued…

Calculate the EntryPoint RVA

- All PE values are stored in RVA format.
  (Relative Virtual Address, an offset from the BaseImage)

The BaseImage is where the application begins in memory.

- RVA EntryPoint = OriginalEntryPoint – BaseImage
  004035B0h - 00400000h = 35B0h

- The Original EntryPoint is 35b0h bytes into the executable!

Objectives of PE Unpacking

Step #4 Continued…

Change the EntryPoint value in the PE header.

- Using a PE editor, such as LordPE/ProcDump we change the executables EntryPoint value to 35b0h.
- If we execute the executable now, it will start executing the unpacked data first, not the PE packer!

Objectives of PE Unpacking

- **The dumped executable image is almost able to run, but it is still missing one vital piece of information.**

- **It does not have a valid Import Address Table!**

- **The current import address table is that of the PE packer itself!**
    - **It only has three entries!**

        LoadLibaryA()

        GetProcAddress()

        ExitProcess()

Remember: The PE-packer uses LoadLibaryA/GetProcAddress to populate the Import Address Table of .PACKED-DATA!

Objectives of PE Unpacking

Step #5: Rebuild the Import Address Table

- **We need to find the Import Address Table of our now unpacked data.**
- **We need Windows to populate our import table with the correct values for each external function at runtime.**
- **Without it, the executable will not run. ☹ and static analysis is also harder.**
- **We will overwrite the PE packers own Import Address Table (which only had three entries) with the correct table.**

Objectives of PE Unpacking

- To do this, we use:
    - ImpRec – Google “ImpRec MackT UCF”

- ImpRec will search the executable image in memory (starting from the OEP value) and should find our Import Address Table.

- We then dump it back to disk.

Objectives of PE Unpacking

- Once we have a copy of the import address table on disk, we re-insert it into the dumped executable.
- Overwriting the old Import Address Table with our “full bodied” table.
- Now when we execute the binary windows will populate the Import Address Table with the correct values, allowing us to use external functions.
- Code execution will start at the unpacked data, and bob’s your uncle!

Demo #1
PE Unpacking - UPX.

- Ok so we know what we want to do, lets do it.

- Notepad.exe
  - Packed with UPX (the Ultimate Packer for eXecutables)
  - UPX can be unpacked with upx.exe –d
  - We will modify this binary though so upx will not recognize it, and fail to unpack it.

  - Ok lets unpack it by hand.

PE Unpacking: Automation

PE Unpacking: Automation.

- That was easy, because UPX is an easy PE packer.
- Although it was easy it still took time.
- Time = Money
- Having to unpack 100 UPX packed binaries would become really tedious.
- Tedious = Not fun!
- Since we now know how UPX works, we should automate the unpacking process.
- So, lets write a quick, automated unpacking script for any UPX packed binary, so we don't have to do this by hand again.

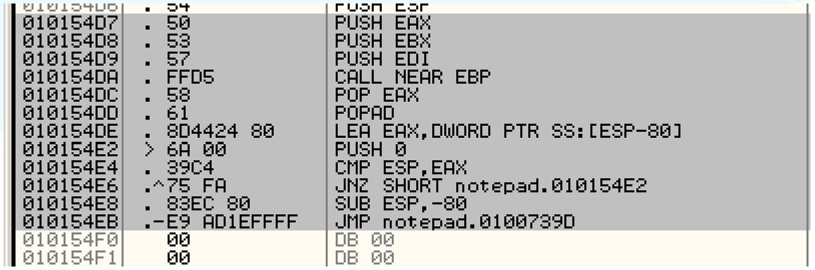
PE Unpacking: Automation.

- To do this will use OllyScript (The scripting language plugin for OllyDBG)
- OllyScript simulates a user's debugging session within OllyDBG.
- We can place breakpoints, step, run, do all the normal OllyDBG tasks, only script them.
- OllyScript is important when dealing with PE packers, it can take hours to unpack a single protector. You don't want to do it too often!
- Learning OllyScript is a <u>must</u> if you plan on doing any unpacking of your own.

PE Unpacking: Automation.

- We know the UPX code flow goes like this

#1 – The target application is un-compressed.
#2 – UPX will populate the packed data's Import Table.
#3 – POPAD
#4 – JMP <OEP>



```
010154D6  . 54            PUSH ESP
010154D7  . 50            PUSH EAX
010154D8  . 53            PUSH EBX
010154D9  . 57            PUSH EDI
010154DA  . FFD5          CALL NEAR EBP
010154DC  . 58            POP EAX
010154DD  . 61            POPAD
010154DE  . 8D4424 80     LEA EAX,DWORD PTR SS:[ESP-80]
010154E2  > 6A 00         PUSH 0
010154E4  . 39C4          CMP ESP,EAX
010154E6  .^75 FA         JNZ SHORT notepad.010154E2
010154E8  . 83EC 80       SUB ESP,-80
010154EB  .-E9 AD1EFFFF   JMP notepad.0100739D
010154F0    00            DB 00
010154F1    00            DB 00
```

The golden rule of UPX is "The first unconditional JMP after POPAD is the OEP JMP"

This works on all versions of UPX, very simple.

PE Unpacking: Automation.

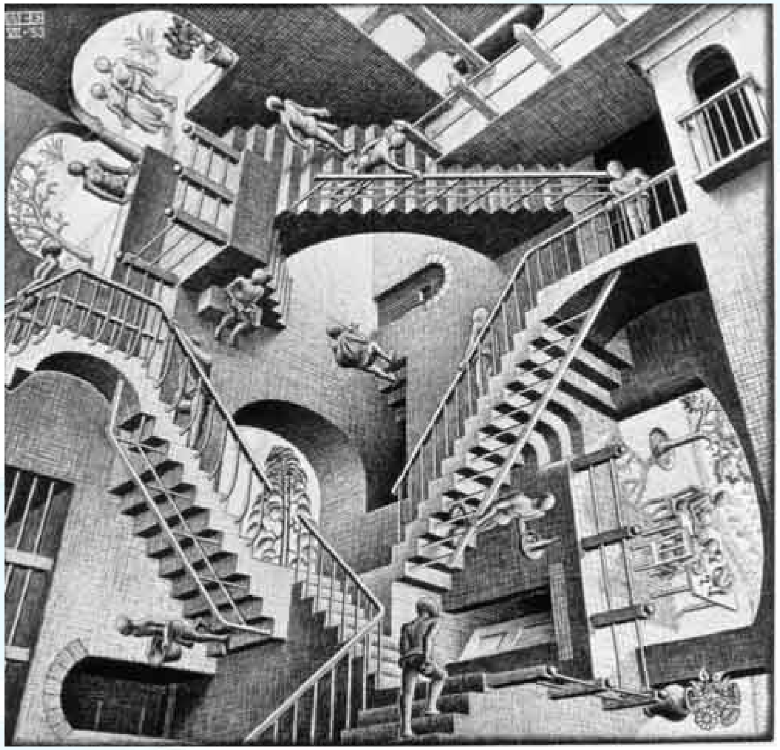
- So we write a script which does something like..
  - Search for the first POPAD in the code.
  - Place a breakpoint on it
  - Run the application

  - Search for the next JMP instruction.
  - Breakpoint again.
  - Run the application again.

  - We end up at the OEP JMP.. Finished.

Demo #2
PE Unpacking: Automation

PE Packers: Getting Tricky.



Getting Tricky.

- So far we have only looked at UPX, a straight-forward relatively friendly PE packer.
- UPX is a great way to learn how PE packers work, in essence all packers are very similar if not identical to UPX.
- But rarely are packers so straight-forward to follow and logical in nature.
- Most PE packers are designed with a focus on anti-unpacking, they don’t want you to unpack them!
- The golden rule still applies though, eventually a packer must execute the unpacked code.
  - Its just a matter of when!

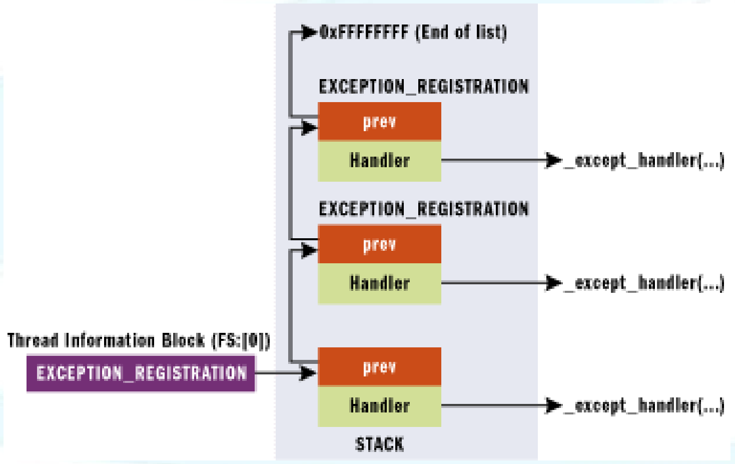
Getting Tricky.

#1 – Exceptions

- Structured Exception Handler – SEH CHAIN
  - A single linked list of exception handlers (frames) to use when an exception occurs.
  - Linked SEH frames make an “SEH chain”
  - Can be used to go from A to B, via an exception.
- The Goal:
  - Create an exception handler to catch an exception.
    - “If you crash, go here”
  - Then raise an exception.
    - “Crash”, and we end up “here”
- The Idea:
  - Get to the exception handler code.
  - Debuggers hate exceptions!

Getting Tricky.

- Each SEH frame within the chain consists of two pointers.
    - A pointer to the previous SEH frame.
    - A pointer to the exception handler for the frame.
- A pointer to the SEH chain is kept inside FS:[0]
    - Thread Information Block - http://en.wikipedia.org/wiki/Win32_Thread_Information_Block
- Real easy to spot, FS:[0] is only used for exception handling.
- Keep an eye on the SEH Chain window inside OllyDBG.



Getting Tricky.

Example:

```
PUSH 00401C84                     ; Push to the stack the new top SE handler location
MOV EAX,DWORD PTR FS:[0]          ; Save the existing top SE frame location to EAX
PUSH EAX                          ; Push the top level back to the stack, its now the 2nd frame
MOV DWORD PTR FS:[0],ESP          ; Move a pointer to the stack into FS:[0] (Chain is now active)
XOR EDX, EDX                      ; Clear EDX
DIV EDX                           ; Divide by zero exception
```

- 00401C84 is now the first SE handler.
- The old SE handler is now the second handler.
- When we execute DIV EDX we will cause an exception.
- This exception will be handled by the top SE handler.

- Tell OllyDBG to Ignore all exceptions.
- Set a breakpoint on 00401C84 (The handler).
- Run the application.
- We end up at the SE handler.

Getting Tricky.

#2 – Detecting a debugger

- Great way to try to stop a pesky reverse engineer is to detect his debugger.
- Windows API Calls.
  - A call to IsDebuggerPresent() will return > 0 if the process is currently running in the context of a debugger, such as OllyDbg.
    - Unable to detect kernel debuggers such as Softice
  - Many other Windows API's can be used to detect a debugger.
    - ZwQueryProcessInformation()
    - CheckIsRemoteDebuggerPresent()
    - SetDebugPrivilege ()

Getting Tricky.

- IsDebuggerPresent API just returns a byte in the PEB.
- The PEB (Process Environment Block) is a process specific area of user land memory which contains details of each running process.
- We find the location of the PEB from the TIB.
  - TIB->PEB->isProcessBeingDebugged

Example:

```
MOV EAX,DWORD PTR FS:[18h]          ; Get the location of the TIB
MOV EAX,DWORD PTR DS:[EAX+30h]  ; Get the location of the PEB
MOVZX EAX,BYTE PTR DS:[EAX+2h]    ; Second byte of PEB = isProcessBeingDebugged.
```

EAX > 0 , debugger is present.

Not hard to spot while tracing.. A “Normal” application would rarely have a need to access FS:[18h]

Getting Tricky.

- Many different ways to defeat debugger checks.
- Automatic plug-ins for OllyDBG
  - HideOD, IsDebuggerPresent
  - These plug-in's hide a debuggers presence by keeping the PEB isBeingDebugged flag at 0
  - They use other methods to hide from each of the debugger detection API's in Windows.
- Ollyscript: 'DBH' – Hide debugger
- Using the OllyDBG command line plug-in
  - "set byte ptr ds:[fs:[30]+2]] = 0"
- Manually patch your way out…

Demo #3
SEH Wonderland

Demo #4
Almost finished!!

- No presentation at a hacker conference is complete without some 0day remote shell!

- Give it up, for an OllyScript bind shell exploit!

- All your debuggers are belong to me!

Conclusion

- PE packing an executable is not hard.
  - Even a modified UPX can thwart static analysis attempts.
- When unpacking an executable, we don't need to know exactly how its payload is packed.
  - All PE packers must unwrap themselves eventually, we can leave this to the PE packer.
- We are simply an observer, until we find the OEP.
- Then dump the executable image to disk.
- Redirect the EntryPoint to the discovered OEP.
- Rebuild the Import Address Table.

- Script it, so we never have to do it again!

Questions ?

http://www.security-assessment.com

paul.craig@security-assessment.com

Links/References

- Iczelion Win32 ASM Page - http://win32assembly.online.fr/
- EXE Tools – http://www.exetools.com
- Yates2k R.E - http://www.yates2k.net/
- Programmers Tools - http://programmerstools.org/
- RETeam.org – http://reteam.org
- ARTeam Research – http://arteam.accessroot.com
- LordPE - http://mitglied.lycos.de/yoda2k/LordPE/info.htm
- TIB - http://en.wikipedia.org/wiki/Win32_Thread_Information_Block
- PEB - http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/PEB.html
- OpenRCE – http://www.openrce.org

- Security-Assessment.com – http://www.security-assessment.com