Process Hollowing

Whereabouts

- What?

  Replace executable code of another process by own code (in memory).

- Why?

  Deceives AV, firewall, IPS. (Well, does it indeed?)
  Privilege escalation possible.
  Makes Reversing harder. (Not really…)

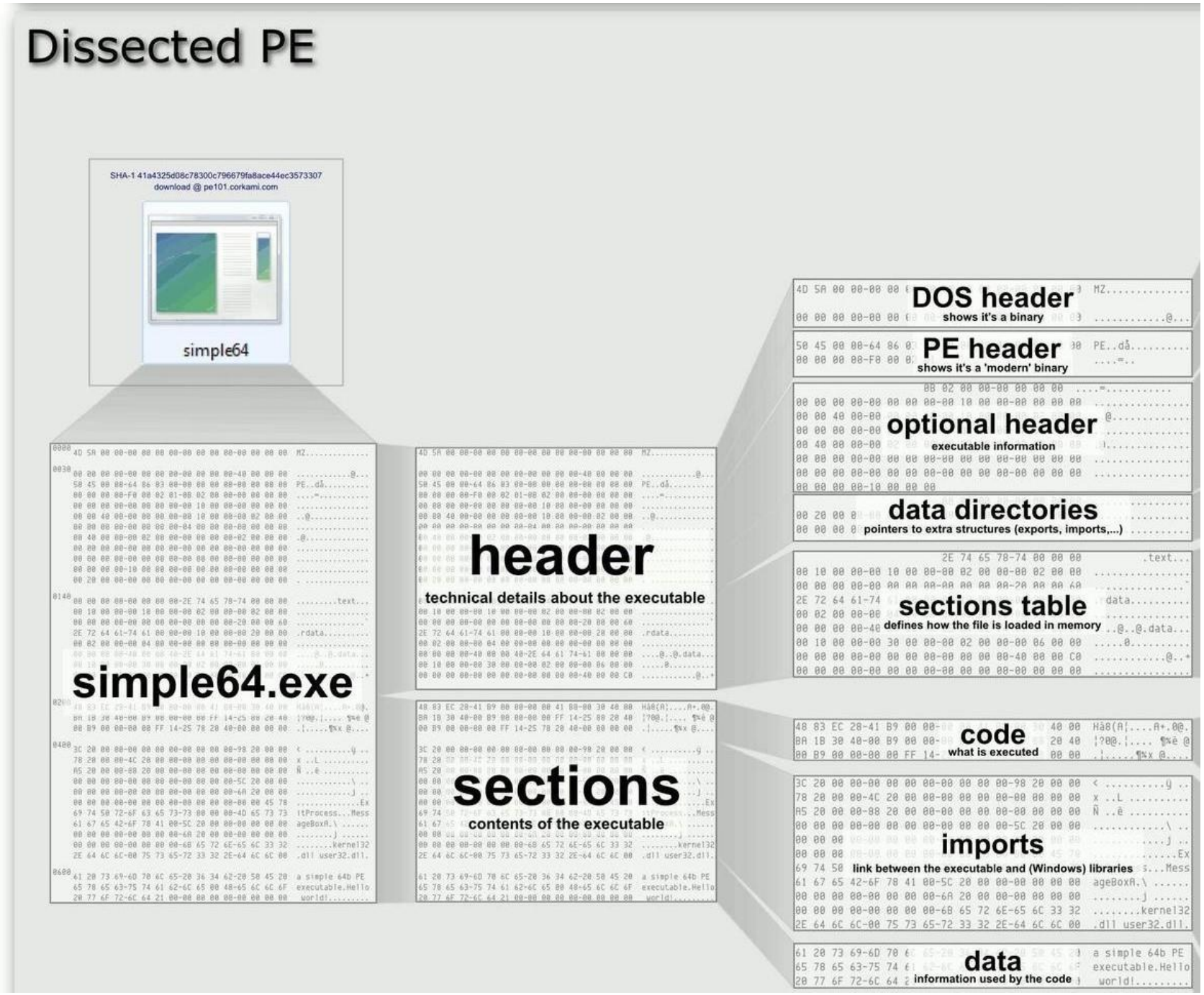
- Who?

  Malware.
  We‘re in the windows world here.

Agenda

- PE file basics
- Windows loader basics
- Process Hollowing: Classic approach
- There are multiple ways
- Injection?
- Detection
- Doppelgäng all the things
- Tool time

Dissected PE




Hexadecimal dump
ASCII dump
Fields
Values
Explanation
DOS header
shows it's a binary
e_magic
e_lfanew
'MZ'
0x40
constant signature
offset of the PE Header ❶
PE header
shows it's a 'modern' binary
Signature
Machine
NumberOfSections
SizeOfOptionalHeader
Characteristics
'PE', 0, 0
0x8664 [AMD 64]
3
0xf0
0x102 [32b EXE]
constant signature
processor: ARM/MIPS/Intel/...
number of sections ❷
relative offset of the section table ❷
EXE/DLL/...
optional header
executable information
Magic
AddressOfEntryPoint
ImageBase
SectionAlignment
FileAlignment
MajorSubsystemVersion
SizeOfImage
SizeOfHeaders
Subsystem
NumberOfRvaAndSizes
0x20b [64b]
0x1000
0x400000
0x1000
0x200
4 [NT 4 or later]
0x4000
0x200
2 [GUI]
16
32 bits/64 bits
where execution starts ❺
address where the file should be mapped in memory
where sections should start in memory ❷
where sections should start on file ❷
required version of Windows
total memory space required
total size of the headers ❸
driver/graphical/command line/...
number of data directories ❹
data directories
pointers to extra structures (exports, imports,...)
ImportsVA
0x2000
RVA* of the imports ❹
sections table
defines how the file is loaded in memory
Sections table
Name
VirtualSize (RVA*)
VirtualAddress (RVA*)
SizeOfRawData (physical size)
PointerToRawData (physical offset)
Characteristics
.text 0x1000 0x1000 0x200 0x200 CODE EXECUTE READ
.rdata 0x1000 0x2000 0x200 0x400 INITIALIZED READ
.data 0x1000 0x3000 0x200 0x600 DATA READ WRITE
For each section, a SizeofRawData sized block is read from the file at PointerToRawData offset.
It will be loaded in memory at address ImageBase + VirtualAddress in a VirtualSize sized block, with specific characteristics.
code
what is executed
x86 assembly
Equivalent C code
sub rsp, 0x28
// stack adjustment
mov r9d, 0
mov r8d, 0x403000
mov edx, 0x40301B
mov ecx, 0
call [0x402088]
MessageBox(0, "Hello World!","a simple PE executable", 0);
mov ecx, 0
call [0x402078]
ExitProcess(0);
imports
link between the executable and (Windows) libraries
Imports structures
descriptors
0x203c
0x205c, 0 (INT*)
0x2098
kernel32.dll
Hint.Name
0,ExitProcess
0x2078
0x205c, 0 (IAT*)
0x204C
0x206a, 0 (INT*)
0x20A5
user32.dll
Hint.Name
0,MessageBoxA
0x2088
0x206a, 0 (IAT*)
0 0 0 0 0
All addresses here are RVAs*
Consequences
after loading,
0x402078 will point to kernel32.dll's ExitProcess
0x402088 will point to user32.dll's MessageBoxA
data
information used by the code
Strings
a simple 64b PE executable\0
Hello world!\0
This is the whole file, however, most PE files contain more elements. Explanations are simplified, for conciseness.
version 1 64b, 25th September 2012

Windows Loader 101

Loading process

❶ Headers

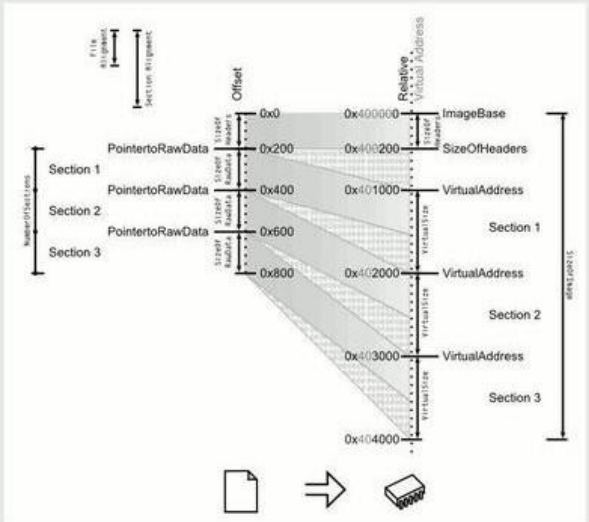
the *DOS Header* is parsed
the *PE Header* is parsed
(its offset is *DOS Header*'s e_lfanew)
the *Optional Header* is parsed
(it follows the *PE Header*)

❷ Sections table

Sections table is parsed
(it is located at: offset (*OptionalHeader*) + *SizeOfOptionalHeader)*
it contains *NumberOfSections* elements
it is checked for validity with alignments:
FileAlignments and *SectionAlignments*

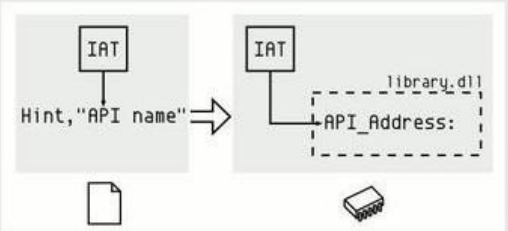
❸ Mapping

the file is mapped in memory according to:
the *ImageBase*
the *SizeOfHeaders*
the Sections table



❹ Imports

DataDirectories are parsed
they follow the *OptionalHeader*
their number is *NumOfRVAAndSizes*
imports are always #2
Imports are parsed
each descriptor specifies a *DLLname*
this DLL is loaded in memory
IAT and *INT* are parsed simultaneously
for each API in *INT*
its address is written in the *IAT* entry



❺ Execution

Code is called at the *EntryPoint*
the calls of the code go via the IAT to the APIs

Process Hollowing: Classic Steps

- Create suspended process
- Unmap destination image from memory
- Allocate new memory in target process
- Copy payload sections into target process
- (Optional: Fix memory protection)
- Set thread context
- Resume target process
- Congratulations!

API Calls

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
 ❶ WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

Different Approach

- Create suspended process (*CreateProcessA)*
- Get image base of suspended process (*ReadProcessMemory)*
- Create two new section objects (*NtCreateSection)*
- Map sections into malware address space (*NtMapViewOfSection)*
- Map payload section also into target process (*NtMapViewOfSection)*
  This section is now shared between the two processes.
- Copy new process image into section 1, copy payload into section 2
- Install jump at section 1 entry point, pointing into payload in section 2
- Map section 1 at base address of target process (*NtMapViewOfSection)*
- Resume target process (*NtResumeThread)*

DLL Injection

- Allocate memory in target process
- Copy name of desired DLL into allocated memory
- Retrieve LoadLibrary from kernel32.dll
- Create and start new thread in target process, which calls LoadLibrary and starts execution on DllMain

⇒Payload delivered without hollowing

⇒But: Noisy, DLL is external dependency

⇒Solution: Reflective DLL Injection – „Do LoadLibrary yourself“

Process Hollowing Detection

- Check for RWX permissions (can easily be fixed by attacker)
- Correlate PE on disk vs. PE in memory
  - Flags all self-modifying processes as (false) positives: packers, self-updaters
- Hook API calls and fuzzy-hash against known sequences
- Other „default“ characteristics:
  - File hashes
  - Hollowing target (svchost.exe, …)
  - Unusual file location / file hidden?

Process Doppelgänging (Black Hat EU ’17)

- Create virtual file in NTFS transaction (*CreateTransaction, CreateFileTransacted)*
- Create PE section in virtual file (*NtCreateSection)*
- Rollback transaction: Section is now only in memory – fileless!
- Create new process from section (*NtCreateProcessEx)*
- Need to do some Windows Loader steps manually:
  - Fix PEB and process parameters
  - Relocation
  - …
- Start process main thread by calling *NtCreateThreadEx*


TOOL
TIME

Current Development

- Input:
  One or multiple .exe file „payloads“.
  Zero or multiple drop files (use case: DLLs).
- Output:
  Single executable „package“ with encrypted payloads and drop files.
- On package execution:
  Decrypt and drop files.
  Decrypt and instanciate payloads:
    One new process per payload, using Process Hollowing.
- Convenience by using complete executables. Use case: standalone ncat reverse shell