SAFE. IN THE DEEP SENSE.

USA 2016

Certificate Bypass: Hiding and Executing Malware from a Digitally Signed Executable

August 2016

Submitted by Deep Instinct Research Team

Executive Summary

Malware developers are constantly looking out for new ways to evade the detection and prevention capabilities of security solutions.

A commonly used technique to achieve those goals is the usage of tools, such as packers and new encryption techniques, making the malicious code unreadable by security products.

If the security solution cannot unpack the compressed or encrypted malicious content (or at least unpack it dynamically), then the security solution will not be able to identify that it is facing malware. However, packed and encrypted files can be identified both on disk by their special characteristics and during file execution when the file content needs to be unpacked or decrypted.

In this paper we present a different evasion technique for malicious code that bypasses security vendors, both on the disk and during loading, by storing the malicious code inside signed files without invalidating the digital signature. It also evades detection during execution time, by using reflective EXE loading of the malicious code. Thus, our technique allows the execution of persistent malicious code to remain hidden from current software solutions.

Introduction

A packer is a tool that transforms an executable into another executable, which exhibits the same or extended functionality. However, it has a different footprint in the file system where it resides.

Currently, real-world packers have been developed for mainly two reasons. The first one is to reduce the size of the executable by compressing data and uncompressing it “on the fly” (i.e. dynamically rather than as the result of statically predefined during execution). These packers, often referred to as compressors, were highly popular in the early days of personal computers when the size of the executable was far more important due to limited storage space.

Second, packers were also developed to make reverse engineering of executables significantly more difficult. These packers are often referred to as “protectors”, as they attempt to protect the original executable from malware researchers. They use obscure methods to prevent straightforward analysis of the executable, often by dynamically detecting common analysis tools, such as debuggers, in various ways. The fact that packers change the footprint on the disk, often mitigating static signature-based techniques as a result, has already been discovered and exploited for quite a while by malware writers for rendering their malicious tools.

In order to create better security solutions against malware and packers, we first had to acknowledge the fact that most of the sophisticated malware are packed in some way. According to research conducted by McAfee Labs [1], titled: “The good, the bad, and the unknown”, up to 80% of malware is obfuscated with packers and compression techniques.

Most malware developers generally use known packers. As a result, packers’ signatures have been adopted by commercial security solutions. Furthermore, those known packers also have unpackers that security solutions might use prior to scanning. However, this approach has the same shortcoming as signatures for malware detection: it is not effective for unknown obfuscation techniques or custom packers. This is also a major limitation, because according to Morgenstern and Pliz [2], 35% of malware are packed by a custom packer.

In order to challenge security solutions and find possible security breaches, we had to find the Achilles’ heel of most of them.

An ordinary custom packer is one option, but it might be identified by two main facts:

1. A packed executable usually contains encrypted or compressed sections, which raises the entropy for the specific section. This is a strong indication for a packed executable, which sometimes is a heuristic for a malicious file.

2. In order to decrypt or decompress the sections mentioned above, a packer must also have a “clean” bootstrap section containing a stub, which opens the encryption and transfers control to the original code (now un-encrypted) section. This means changing the PE headers to match the new sections (as well as changing the entry point). This might also be related to a malicious activity, since its can be considered a deviation from a normal PE file format.

For these reasons, we will present a more sophisticated solution: one that includes hiding a malicious file, having the ability to execute it, and still maintaining a normal PE execution (without encrypting the main sections of the file).

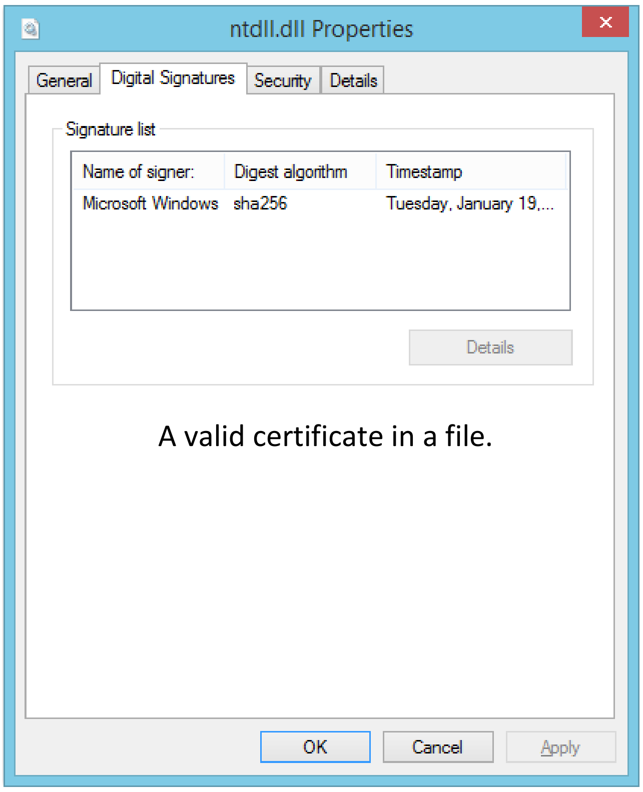
Windows' process of identifying a valid certificate

In order to understand how Windows reads the certificate section of files, we first need to dive into the PE header structure and understand the meaning of the relevant fields:

IMAGE_OPTIONAL_HEADERS

This header contains information about the logical layout in the PE file. There are 31 fields in this structure. We will explain only those fields that are relevant for our purposes:

- Checksum
  The image file checksum. The following files are validated at load time: all drivers, any DLL loaded at boot time, and any DLL loaded into a critical system process [MSDN].
- DataDirectory
  An array of IMAGE_DATA_DIRECTORY structures. Each structure gives the RVA of an important data structure in the PE file, such as the import address table.



Name of signer:	Digest algorithm	Timestamp
Microsoft Windows	sha256	Tuesday, January 19,...

A valid certificate in a file.

IMAGE_DATA_DIRECTORY

Since this structure gives the RVA of important data structures, we will only focus on the content of IMAGE_DIRECTORY_ENTRY_SECURITY (Information about the Attribute Certificate Table).

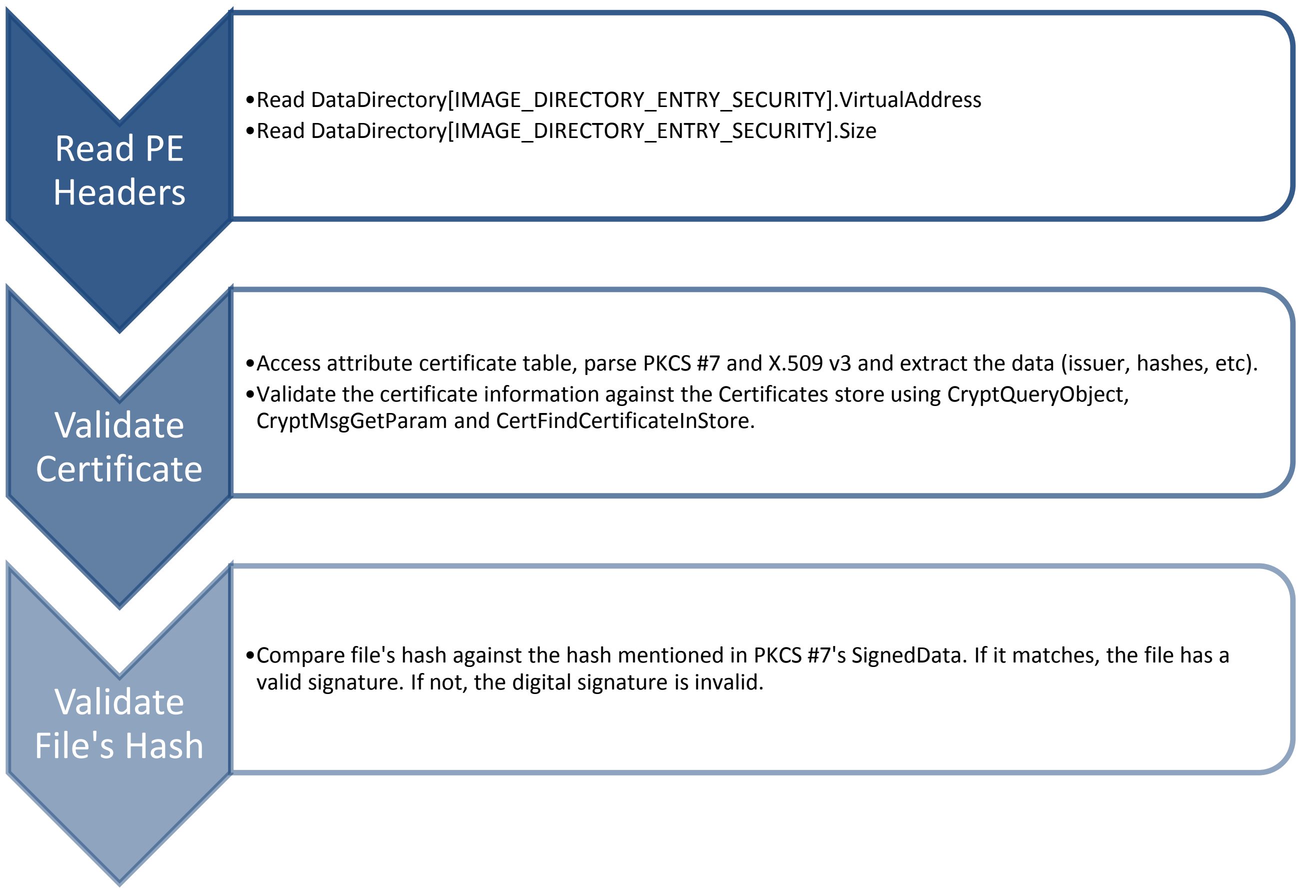
IMAGE_DATA_DIRECTORY item contains two members:

1. VirtualAddress – the relative virtual address to the appropriate table.
2. Size – the size of the table, in bytes.

Windows reads these members and knows the location and size of the attribute certificate table. The size is also mentioned at the beginning of the attribute certificate table (as dwLength member in the WIN_CERTIFICATE structure, which is an alias for the attribute certificate table) and it is something we should consider as well.
Windows is using Authenticode in order to determine the origin and integrity of software binaries, and it is based on Public-Key Cryptography Standards (PCKS) #7 while using X.509 v3 certificates to bind an Authenticode-signed binary to the identity of a software publisher. In order to validate the integrity of the file and make sure it hasn't been tampered with, it calculates the hash (excluding 3 fields – Checksum, IMAGE_DIRECTORY_ENTRY_SECURITY entry in the DataDirectory, and the attribute certificate table itself) and compare the result against the hash mentioned in the

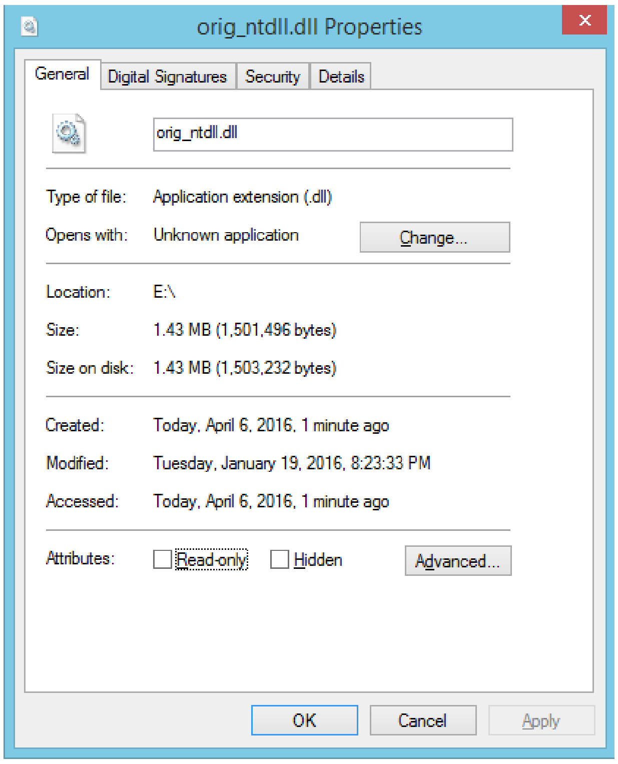
SignedData structure in PKCS #7 (if the two are different, the code has been changed, and the digital signature becomes invalid).

The process of validating PE's signature:


Read PE Headers
•Read DataDirectory[IMAGE_DIRECTORY_ENTRY_SECURITY].VirtualAddress
•Read DataDirectory[IMAGE_DIRECTORY_ENTRY_SECURITY].Size
Validate Certificate
•Access attribute certificate table, parse PKCS #7 and X.509 v3 and extract the data (issuer, hashes, etc).
•Validate the certificate information against the Certificates store using CryptQueryObject, CryptMsgGetParam and CertFindCertificateInStore.
Validate File's Hash
•Compare file's hash against the hash mentioned in PKCS #7's SignedData. If it matches, the file has a valid signature. If not, the digital signature is invalid.

Because Windows excludes the fields mentioned above from the hash calculations, we can inject data to the certificate table without damaging the validity of the file's certificate.

By appending malicious content to the end of the certificate table and modifying the relevant fields accordingly (Size [Both in DataDirectory[IMAGE_DIRECTORY_ENTRY_SECURITY] and in WIN_CERTIFICATE] and CheckSum), we can modify the file without harming the validity of the certificate.

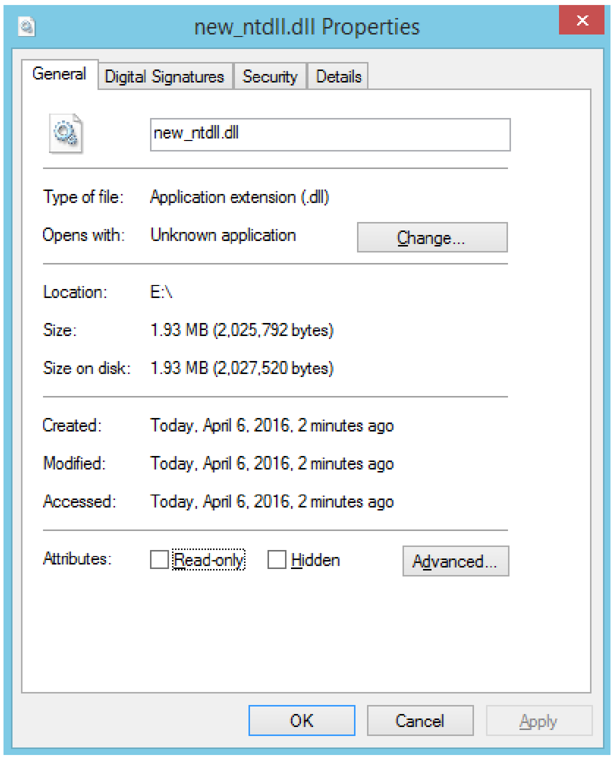


orig_ntdll.dll Properties

General | Digital Signatures | Security | Details

orig_ntdll.dll

Type of file: Application extension (.dll)
Opens with: Unknown application
Location: E:\
Size: 1.43 MB (1,501,496 bytes)
Size on disk: 1.43 MB (1,503,232 bytes)
Created: Today, April 6, 2016, 1 minute ago
Modified: Tuesday, January 19, 2016, 8:23:33 PM
Accessed: Today, April 6, 2016, 1 minute ago
Attributes: ☐ Read-only ☐ Hidden



new_ntdll.dll Properties

General | Digital Signatures | Security | Details

new_ntdll.dll

Type of file: Application extension (.dll)
Opens with: Unknown application
Location: E:\
Size: 1.93 MB (2,025,792 bytes)
Size on disk: 1.93 MB (2,027,520 bytes)
Created: Today, April 6, 2016, 2 minutes ago
Modified: Today, April 6, 2016, 2 minutes ago
Accessed: Today, April 6, 2016, 2 minutes ago
Attributes: ☐ Read-only ☐ Hidden

On the left: the original ntdll.dll with a valid Digital Signature.

On the right: a modified ntdll.dll, with a valid Digital Signature and malicious content embedded into the certificate table.

All the security solutions we have checked did not recognize the file as a malicious one even when the malicious file was not encrypted. Since the embedded malware is not part of the execution process, those solutions did not identify the malicious content even upon execution of the signed file.

This way, we are able to hide malicious content in files across windows file system, without being identified.

Reflective PE Loader

Having a malicious file in the disk without having it identified is nice, but having nothing to do with it makes it less interesting. That is the reason why we wrote a Reflective PE Loader: to execute PE files directly from memory.

Since the PE execution process in Windows is undocumented, we researched how to be able to replicate the PE execution process on our own.

The definition of two key terminologies:

Payload PE: the executable, which we will execute in memory.

Host Process: the executable, which will execute Payload PE in its memory space.

In order to execute the Payload PE in Windows, we need to take the following steps for the Host Process and the Payload PE:

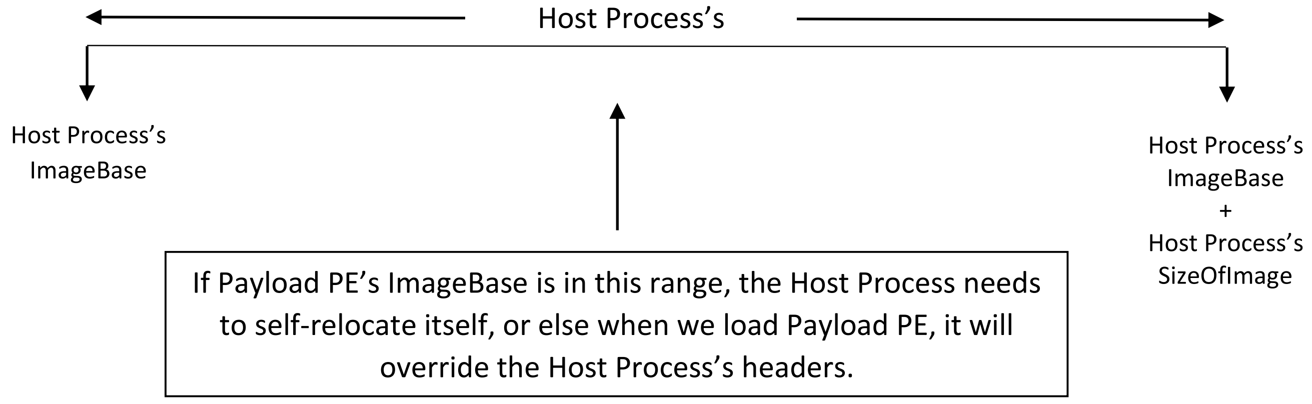
The steps required for preparing the Host Process:

1. Extract pointers to three structures (two from the Payload PE, and one from the Host Process):
   - IMAGE_DOS_HEADER: the first structure of the PE file. It contains global information about the program file and how to load it. Most of the information contained in this structure was relevant to DOS software, and is only supported for backwards compatibility.
   - IMAGE_NT_HEADERS: contains information about the logical layout in the PE file.
   - PEB (Process Environment Block) of the Host Process: when Windows' kernel loads an executable image in memory, it automatically builds a PEB for it. Most of this structure is undocumented. For our purposes, we only need the Image Base Address member.

2. Validating target PE: we do this by reading the Signature field from IMAGE_NT_HEADER and comparing it with 0x4550 (PE\0\0), as well as reading Magic member from IMAGE_OPTIONAL_HEADER and comparing it with 0x010b (IMAGE_NT_OPTIONAL_HDR32_MAGIC).

3. Since we execute the Payload PE inside the Host Process' address space, we need to check if the Host Process needs to self-relocate itself so that Payload PE will be fit inside of it. To verify this, we check if Payload PE's ImageBase address is in the address space where the Host Process resides. We compare the following two elements (both have to be true in order for self-relocation to occur):

1. The ImageBase address of the Payload PE is bigger than the ImageBase address of the Host Process; and
2. The ImageBase address of the Payload PE is smaller than the Host Process's ImageBase + Host Process's SizeOfImage (where SizeOfImage is the size of the image, in bytes, including all headers).


Host Process's
Host Process's ImageBase
Host Process's ImageBase + Host Process's SizeOfImage
If Payload PE's ImageBase is in this range, the Host Process needs to self-relocate itself, or else when we load Payload PE, it will override the Host Process's headers.

In order to self-relocate the Host Process, we allocate (using VirtualAlloc with PAGE_EXECUTE_READWRITE and MEM_COMMIT | MEM_RESERVE as the allocation type) enough space to contain the Host Process (we know the size by reading OptionalHeader.SizeOfImage).

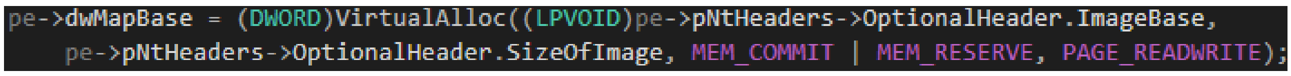
When the memory block is allocated, we relocate the Host Process from the old address, to the new address. This includes:

- Copy the content of the Host Process's image to the new address returned by VirtualAlloc
- Update function addresses by processing the IAT (see below for more information about this process).
- Apply image relocation (by reading the Host Process's Base Relocation entry).
  Because we move the PE image to another base address, we need to relocate data in the PE to match the new base address so that they are correct again. Before processing the relocation table, we first need to make sure that the file has a relocation directory.
  Base relocation just contains a list of locations in the image that need a value added to them.
  For example, if a DWORD contained the address 0x900000 and the PE ImageBase address was 0x400000, but it was actually loaded to the address 0x500000, we calculate the new value for the above DWORD:
  0x900000 – 0x400000 + 0x500000 = 0xA00000

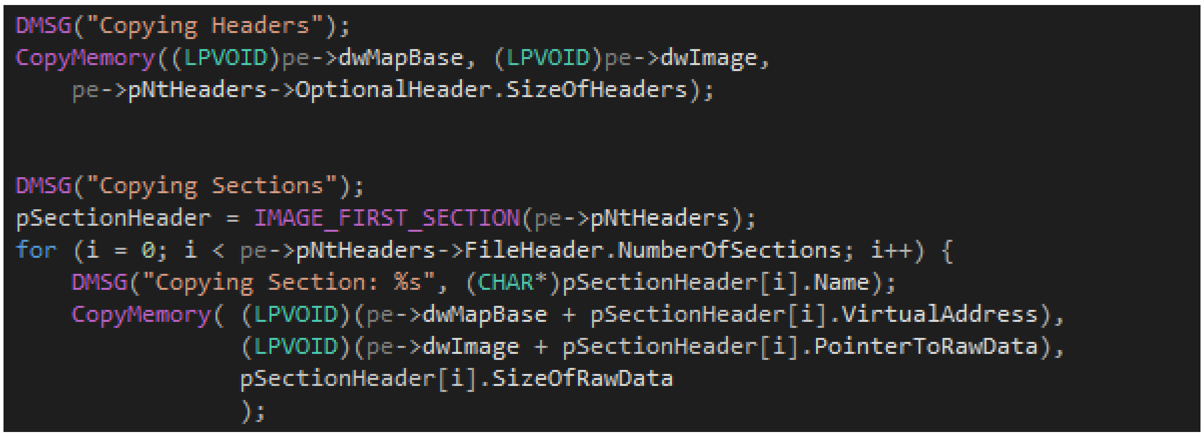
After finishing the entire relocation process, we jump to a pre-calculated offset. The offset is a pointer to a function, which takes all the necessary steps before executing the Payload PE. This offset is also relative to the image base address, of course.

The steps required for preparing the Payload PE:

1. **Mapping the Image**
   Before jumping to the entry point address, we need to map the Payload PE to the memory and copy all the relevant headers.
   In order to map the image in the right way, we first need to extract the Payload PE’s ImageBase and allocate sufficient space to contain the image. We do this using VirtualAlloc, allocating memory at a specific memory address (OptionalHeader->ImageBase).
   Upon successful initialization of the memory, we copy the headers and the sections of the Payload PE. We determine the size to copy by looking at OptionalHeader->SizeOfHeaders, as well as by going section after section and copying it to the newly allocated memory (we determine the number of sections by looking at the FileHeader->NumberOfSection member).



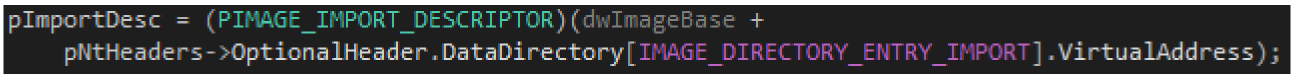
```
pe->dwMapBase = (DWORD)VirtualAlloc((LPVOID)pe->pNtHeaders->OptionalHeader.ImageBase,
    pe->pNtHeaders->OptionalHeader.SizeOfImage, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```



```
DMSG("Copying Headers");
CopyMemory((LPVOID)pe->dwMapBase, (LPVOID)pe->dwImage,
    pe->pNtHeaders->OptionalHeader.SizeOfHeaders);


DMSG("Copying Sections");
pSectionHeader = IMAGE_FIRST_SECTION(pe->pNtHeaders);
for (i = 0; i < pe->pNtHeaders->FileHeader.NumberOfSections; i++) {
    DMSG("Copying Section: %s", (CHAR*)pSectionHeader[i].Name);
    CopyMemory( (LPVOID)(pe->dwMapBase + pSectionHeader[i].VirtualAddress),
                (LPVOID)(pe->dwImage + pSectionHeader[i].PointerToRawData),
                pSectionHeader[i].SizeOfRawData
                );
```

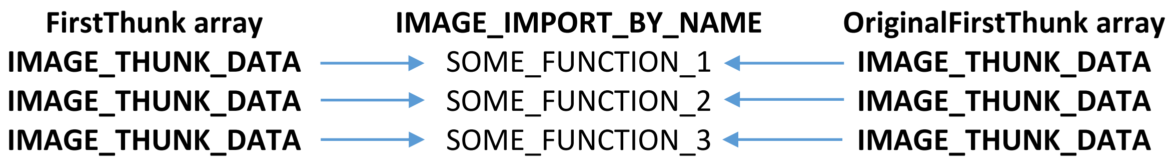
2. **Processing the IAT**
   In order for the new PE to work properly, we need to load its imports on our own.
   This is a fairly straightforward process: we first need to determine the address where the IAT lays in the file (IMAGE_IMPORT_DESCRIPTOR). We do this by reading the VirtualAddress member from DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT] (please see above for further explanation about DataDirectory structure). Since the Virtual Address is the offset from the image base, in order to get the actual memory address, we need to append VirtualAddress to the ImageBase address:



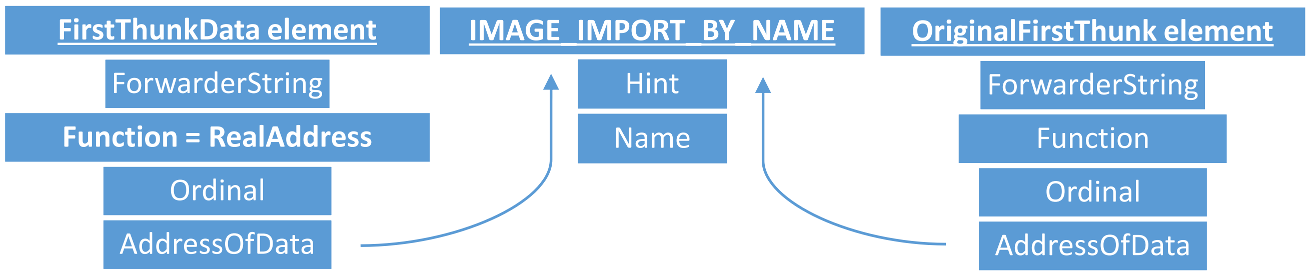
```
pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)(dwImageBase +
    pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
```

The important parts of an IMAGE_IMPORT_DESCRIPTOR are:

1. Name field: contains the DLL name we need to load.
2. OriginalFirstThunk: contains an RVA to an IMAGE_THUNK_DATA array. Each element in the array represents a function whose address we need to retrieve.
3. FirstThunk: points to another array (which is the exact duplicate) of IMAGE_THUNK_DATA.



A question that arises: why do we need two arrays which are exactly the same?
When the PE file is loaded into memory, the PE loader retrieves the addresses of all the imported functions, and then it assigns the "Function" field in each IMAGE_THUNK_DATA element (the one pointed by FirstThunk) with the real address of the function it just received. Upon execution, FirstThunk elements will now change to:



Traversing through pImportDesc will reveal all the DLL names that we need to load to let the PE work properly. We do this by reading the Name field and by using LoadLibrary, we load the DLL. Since loading the DLL is insufficient, we also need to retrieve function's addresses.
Each element of IMAGE_IMPORT_DESCRIPTOR contains a member named OriginalFirstThunk, which is an offset of an IMAGE_THUNK_DATA union. This union contains a member named "AddressOfData", which is a pointer to a IMAGE_IMPORT_BY_NAME structure. Because some functions are exported by ordinal only, we need to determine how to treat the function (import by address / import by ordinal).
To check this, we use a handy constant (IMAGE_ORDINAL_FLAG, which contains the value 0x80000000 for 32bit systems, and 0x8000000000000000 for 64 bit) and use it to check for the Most Significant Bit in the Ordinal field in the current IMAGE_THUNK_DATA we are looking at. For example, if a function is exported by ordinal and its ordinal is 0x4321, then the Ordinal field will contain the value 0x80004321. In case we need to import by name, we access the "Name" field in IMAGE_IMPORT_BY_NAME (Pointed by AddressOfData) and use GetProcAddress to retrieve the function's address.
When we successfully retrieve the address, we assign it to the "Function" field in the current IMAGE_THUNK_DATA (Pointed by FirstThunk, as explained above).
The number of IMAGE_THUNK_DATA elements in FirstThunk and OriginalFirstThunk depends on the number of function the PE file imports from the DLL. If the PE file imports ten functions from kernel32.dll, then the "Name" field in IMAGE_IMPORT_DESCRIPTOR will contain the RVA of the string "kernel32.dll", and there will be ten IMAGE_THUNK_DATAs in FirstThunk and OriginalFirstThunk.

- Please bear in mind, Borland compilers do not produce OriginalFirstThunk (the value will be 0).
- We know we reached the end of IMAGE_THUNK_DATA array by checking "AddressOfData" field (NULL means we reached the end of the array).
- We know we reached the end of IMAGE_IMPORT_DESCRIPTOR by checking the "Name" field (NULL means we reached the end of the array).

With all the basic concepts clear, the next step in the process is to retrieve the virtual address of each function used. To do that, we go through IMAGE_IMPORT_DESCRIPTORs array, and for each element, we go through its IMAGE_THUNK_DATAs elements. In order to retrieve the function's address, we previously used LoadLibrary and got the base address of the DLL in memory.

At this stage, we are able to retrieve the address using two methods:

1. Retrieve by ordinal number
   Going through the DLL's export table (IMAGE_EXPORT_DIRECTORY) OptionalHeader->DataDirectory[IMAGE_DIRECTORY_EXPORT] and calculating the offset for the appropriate function, according to the ordinal.

```
if (IMAGE_ORDINAL_FLAG & pThunkDataOrig->u1.Ordinal) {
    PIMAGE_DOS_HEADER       _dos;
    PIMAGE_NT_HEADERS       _nt;

    _dos = (PIMAGE_DOS_HEADER)hDllModule;
    _nt = (PIMAGE_NT_HEADERS)(((DWORD)hDllModule) + _dos->e_lfanew);

    pExportDir = (PIMAGE_EXPORT_DIRECTORY)
        (((DWORD)hDllModule) + _nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    pFuncRva = (((DWORD)hDllModule) + pExportDir->AddressOfFunctions) + (((IMAGE_ORDINAL(pThunkDataOrig->u1.Ordinal) - pExportDir->Base)) * sizeof(DWORD));
    pFuncRva = ((DWORD)hDllModule) + *((DWORD*)pFuncRva);

    pThunkData->u1.Function = pFuncRva;
```

2. Retrieve by name, using GetProcAddress.

Executing Payload PE

Once everything is in place in memory, we need to take three actions in order to execute the file:

1. Change the protection on a region of committed pages in the virtual address space of the Payload PE. We change the permission accordingly, in order to be able to execute it from memory:

```
if (!VirtualProtect(    (LPVOID)pe->dwMapBase, pe->pNtHeaders->OptionalHeader.SizeOfImage,
                        PAGE_EXECUTE_READWRITE, &dwOld)) {
    return FALSE;
}
```

2. Fix the ImageBase address in the PEB (this step is unnecessary. We need to change the address of the ImageBase in case the Payload PE is using this address to calculate different offsets into the file, such as Zeus Trojan):

```
peb = (_PPEB)__readfsdword(0x30);
peb->lpImageBaseAddress = (LPVOID)pe->dwMapBase;
```

3. Set the EntryPoint address and execute the file:

```
dwEP = pe->dwMapBase + pe->pNtHeaders->OptionalHeader.AddressOfEntryPoint;
DMSG("AddressOfEntryPoint VA: 0x%08x, EntryPoint Address: 0x%08x", pe->pNtHeaders->OptionalHeader.AddressOfEntryPoint, dwEP);
__asm {
        call dwEP
}
```

When executed from the Host Process, Payload PE will be executed from the host process memory space and thus, not be visible to tools such as Windows Task Manager or security software.

Summary

As proven in this paperwork, we are able to hide malicious content in signed files without harming the actual certificate of the file. Our tests on leading security solutions have shown that none of them identified the digitally signed file as malicious, **even when the malicious content was not encrypted**.

Since we do not have the ability to execute code from the certificate section, we demonstrated in this whitepaper that we could write a functional Reflective PE Loader, which will execute PE files directly from memory without any remarks and traces on disk or running processes.

Since this Reflective PE Loader is still considered as a working POC, we faced three limitations:

1. We do not support 64 bit.
2. We do not support DLL Forwarding (in a recent test we ran on over 10,000 samples of malware, none of them were using DLL Forwarding).
3. Upon closing Payload PE, the Host Process will also be closed since Payload PE is using ExitProcess. The solution will be to hook ExitProcess and redirect its execution.

Malware developers and hackers are constantly searching for advanced techniques to bypass security solutions by steering away from the classic structure of packers where everything is located in one file. This includes finding ways that are not dependent on each other and connecting them. By adopting an attacker’s mindset, the security industry can creatively identify attack vectors and flaws, offering better protection.

References

[1] McAfee Labs – a whitepaper: “The good, the bad, and the unknown”, 2011

http://www.techdata.com/mcafee/files/MCAFEE_wp_appcontrol-good-bad-unknown.pdf

[2] Maik Morgenstern and Hendrik Pilz – “Useful and useless statistics about viruses and anti-virus programs”, 2010

https://www.av-test.org/fileadmin/pdf/publications/caro_2010_avtest_presentation_useful_and_useless_statistics_about_viruses_and_anti-virus_programs.pdf

[3] Matt Pietrek - Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

https://msdn.microsoft.com/en-us/library/ms809762.aspx

Deep Instinct LTD.
23 Derech Menahem Begin, Tel Aviv, 67021 Israel +972 (3) 545-6600
501 Folsom Street, Suite 400, San Francisco, CA 94105, USA +1 (855) 522-2223
www.deepinstinct.com