PErfidious

Make PE Backdooring Great Again!

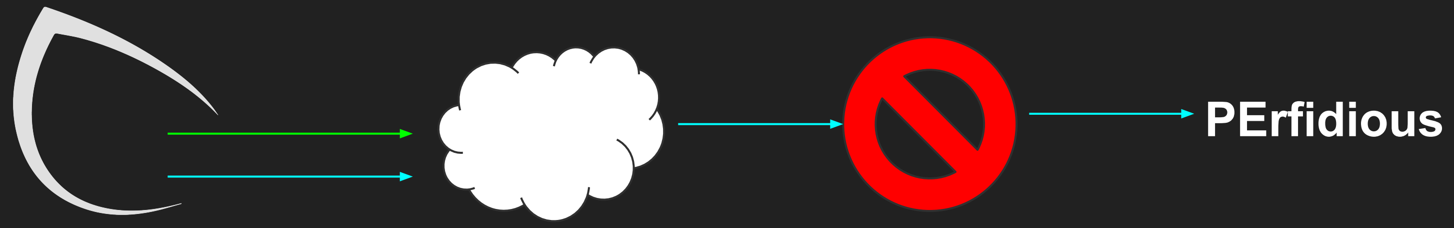
tinyurl.com/hitb2019

About Me

- My name is Shreyans Devendra Doshi
- Cybersecurity Graduate Student @ UMD
- Graduate Teaching Assistant (Reverse Software Engineering) @ UMD
- Previously worked as a Malware Research Intern @ Cybrary Inc.
- Like reverse engineering and malware analysis.

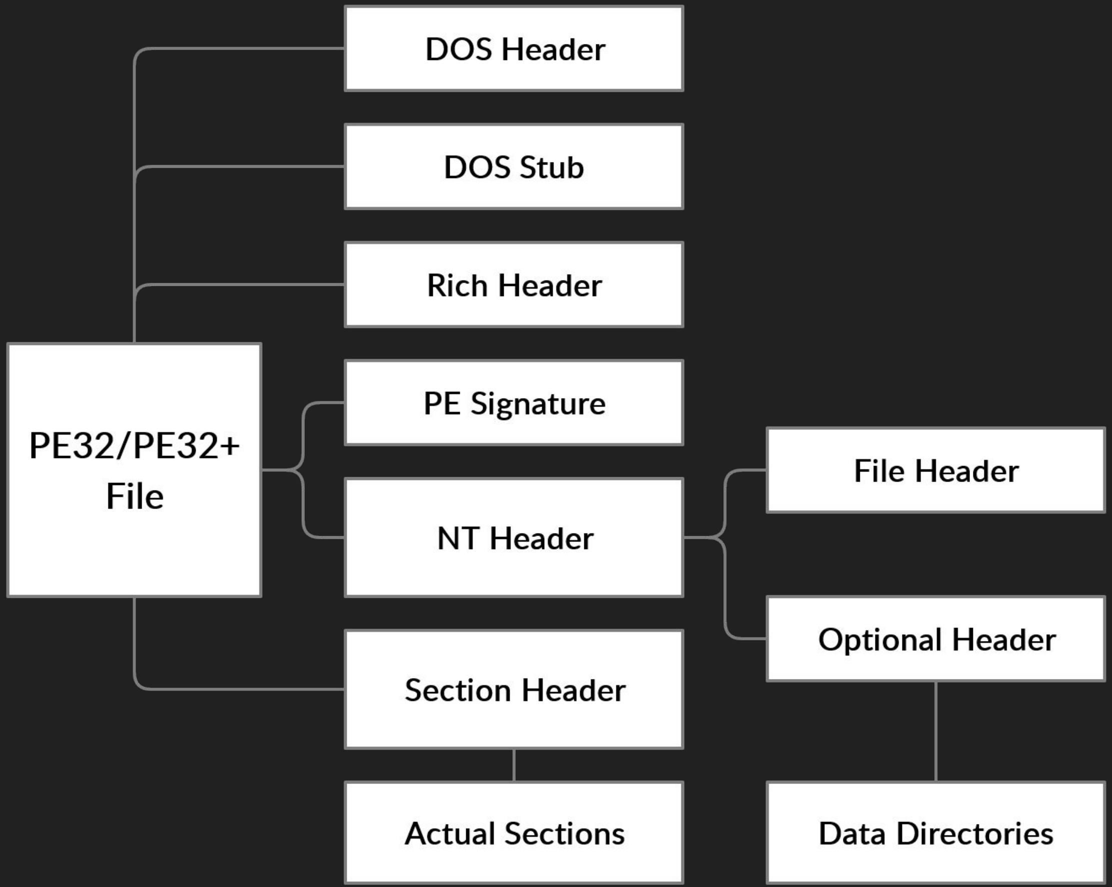
@0xbuilder @0xbuilder

Context



PE File Format

Components of a PE File



DOS Header (64 bytes)

```
struct _IMAGE_DOS_HEADER {
        WORD  e_magic          ← MZ Header signature
        WORD  e_cblp           ← Bytes on last page of file
        WORD  e_cp             ← Pages in file
        WORD  e_crlc           ← Relocations
        WORD  e_cparhdr        ← Size of header in paragraphs
        WORD  e_minalloc       ← Minimum extra paragraphs needed
        WORD  e_maxalloc       ← Maximum extra paragraphs needed
        WORD  e_ss             ← Initial (relative) SS value
        WORD  e_sp             ← Initial SP value
        WORD  e_csum               ← Checksum
        WORD  e_ip             ← Initial IP value
        WORD  e_cs             ← Initial (relative) CS value
        WORD  e_lfarlc         ← File address of relocation table
        WORD  e_ovno           ← Overlay number
        WORD  e_res[4]         ← Reserved words
        WORD  e_oemid          ← OEM identifier (for e_oeminfo)
        WORD  e_oeminfo        ← OEM information (Specific to e_oemid)
        WORD  e_res2[10]       ← Reserved words
        DWORD e_lfanew         ← Offset to extended header
}
```

DOS Stub (Variable)

```
struct DOS_STUB {
        VAR message                  ← '$' terminated string

OR

        DOS Program                  ← The stub can contain an entire DOS program
}
```

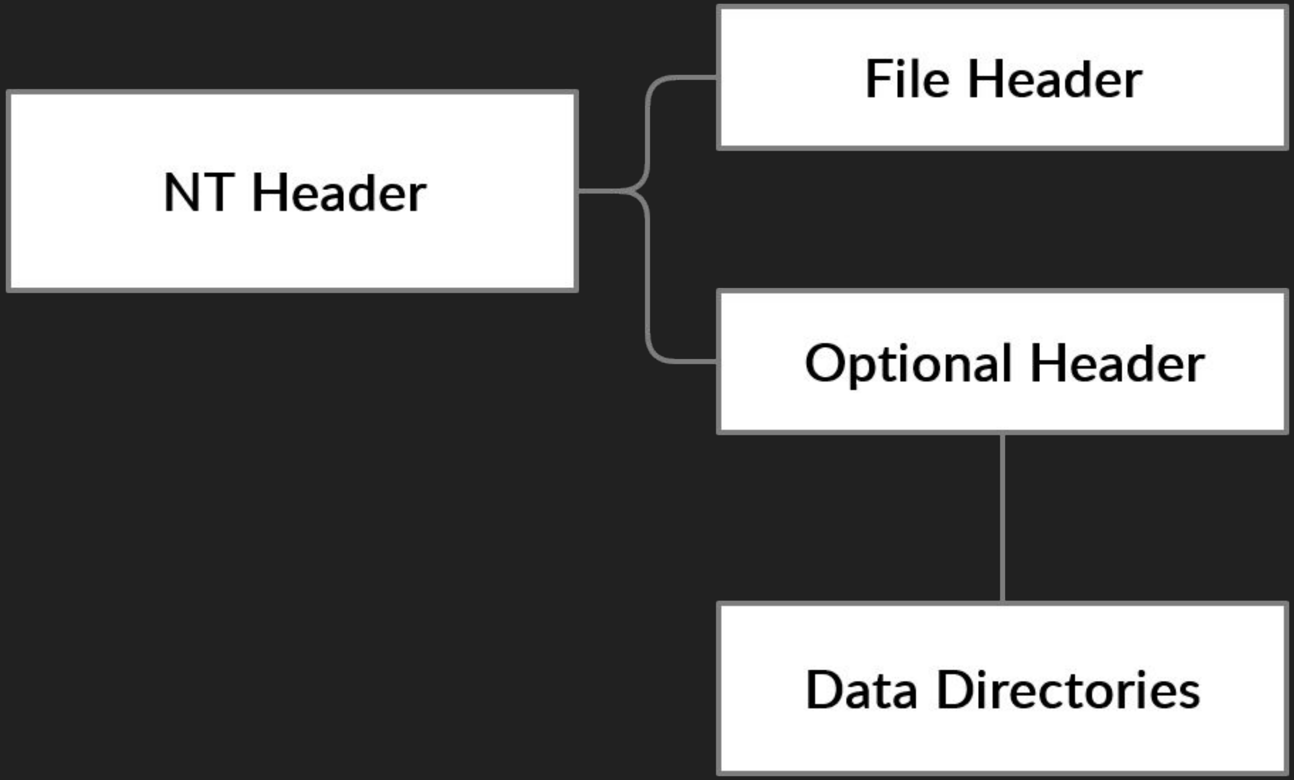
Rich Header (Variable)

```
struct RICH_HEADER {
        WORD DanS_ID                      ← DanS ID = checksum(\x53\x6e\x61\x44) (SnaD)
        WORD  Checksum Padding 1          ← checksum(\x00\x00\x00\x00)
        WORD  Checksum Padding 2          ← checksum(\x00\x00\x00\x00)
        WORD  Checksum Padding 3          ← checksum(\x00\x00\x00\x00)

        DWORD CompID
                | → Count       [0:4]
                | → ProductID   [4:6]
                | → BuildID     [6:8]
            .
            .
            .
        n CompIDs

        WORD RichID                       ← DanS ID = checksum(\x52\x69\x63\x68) (Rich)
        WORD Checksum                     ← The actual checksum value
        VAR GarbageData
}
```

PE Signature

WORD → \x50\x45\x00\x00 → PE\0\0

NT Header (Variable)



File Header (22 bytes)

```
struct _IMAGE_FILE_HEADER {
        WORD  Machine                  ← Machine Type
        WORD  NumberOfSections         ← Number of sections in the PE file
        DWORD TimeDateStamp            ← Time from January 1st 1970, 00:00:00
        DWORD PointerToSymbolTable     ← RVA to the Symbol Table
        DWORD NumberOfSymbols          ← Total number of symbols in the Symbol table
        WORD  SizeOfOptionalHeader     ← Size of the Optional Header
        WORD  Characteristics          ← Characteristics of the PE file
}
```

Optional Header (100+/116+ bytes)

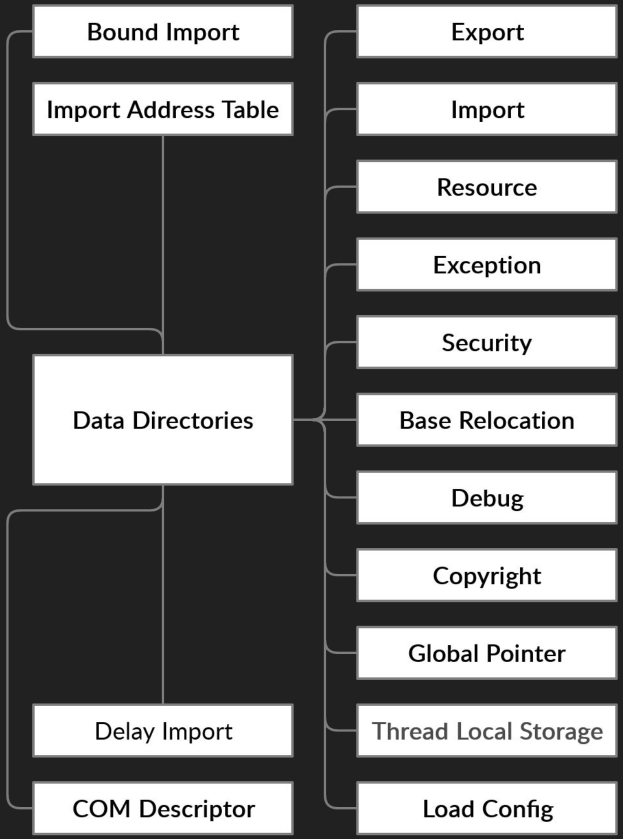
```
struct _IMAGE_OPTIONAL_HEADER32/64 {
        WORD/WORD  Magic
        BYTE/BYTE MajorLinkerVersion
        BYTE/BYTE MinorLinkerVersion
        DWORD/DWORD SizeOfCode
        DWORD/DWORD SizeOfInitializedData
        DWORD/DWORD SizeOfUninitializedData
        DWORD/DWORD AddressOfEntryPoint
        DWORD/DWORD BaseOfCode
        DWORD/QWORD ImageBase
        DWORD/DWORD SectionAlignment
        DWORD/DWORD FileAlignment
        WORD/WORD MajorOperatingSystemVersion
        WORD/WORD MinorOperatingSystemVersion
        WORD/WORD MajorImageVersion
        WORD/WORD MinorImageVersion
        WORD/WORD MajorSubsystemVersion
        WORD/WORD MinorSubsystemVersion
```

...continued

Optional Header (100+/116+ bytes)

```
struct _IMAGE_OPTIONAL_HEADER32/64 {
        DWORD/DWORD Win32VersionValue              ←
        DWORD/DWORD SizeOfImage                    ←
        DWORD/DWORD SizeOfHeaders                  ←
        DWORD/DWORD CheckSum                       ←
        WORD/WORD Subsystem                        ←
        WORD/WORD DllCharacteristics               ←
        DWORD/QWORD SizeOfStackReserve             ←
        DWORD/QWORD SizeOfStackCommit              ←
        DWORD/QWORD SizeOfHeapReserve              ←
        DWORD/QWORD SizeOfHeapCommit               ←
        DWORD/DWORD LoaderFlags                    ←
        DWORD/DWORD NumberOfRvaAndSizes            ←

        DATA DIRECTORIES[n]
}
```

Data Directories



Current Code Injection Techniques

Custom Section Addition

1. Create a custom section containing malicious code.
2. Append this section(mostly at the end) to the PE file.
3. Append the section header and make an entry for the newly added section.
4. Give the section execute permissions in the section header.
5. Change the entry point of the code in the Optional Header to point to the beginning of the newly added section.

Disadvantages of this approach

1. Very easy to detect for endpoint detection systems.
2. Very difficult to do it **CORRECTLY**.
3. Ratio of **stealth gained** v/s **time required for correct implementation** is way too low.

PE Code Caving

1. Find all the code-caves that exist inside the PE file.
2. Out of those code-caves, find all the code-caves that exist inside section(s) with execute permissions.
3. Replace the nulls inside the code-cave with malicious code.
4. Change the AddressOfEntryPoint in the OptionalHeader of the PE file to point to the newly filled code-cave.

Disadvantages of this approach

1. Dependent on finding code-caves inside the PE file.
2. Dependent on finding a code-cave that has execute permissions because altering section permissions is highly susceptible to detection.
3. Dependent on finding malicious code that can fit inside the code-cave.

Why not just edit the .text section?

Adding malicious code to the .text section

1. Use PErfidious to fingerprint the PE file and convert it into a class based structure.
2. Use a function to directly input the malicious code.
3. PErfidious extracts the .text section of the PE file and combines it with the malicious code, thus creating a new .text section with malicious implants.
4. Make changes to the PE file to accommodate the new .text section.

Advantages of this approach

1. Relatively difficult to detect if done right.
2. The malicious code is split into smaller pieces so more difficult to detect.
3. All the other parts of the PE file are left unchanged, so the entropy of the PE file remains relatively unchanged.

How would you detect such an injection?

1. Only allow whitelisted software samples with verified checksum values to run on the machine.
2. Perform graph hash analysis

DEMO TIME

Future of the project

Thank you for your time

Question Time!