An unexpected journey into Microsoft Defender's signature World

retooling.io/blog/an-unexpected-journey-into-microsoft-defenders-signature-world

a b June 28, 2024

Jun 28

Introduction

Microsoft Defender is the endpoint security solution preinstalled on every Windows machine since Windows 7. It's a fairly complex piece of software, addressing both EDR and EPP use cases. As such, Microsoft markets two different products. Microsoft Defender for Endpoint is a cloud based endpoint security solution that combines sensor capabilities with the advantages of a cloud processing. Microsoft Defender Antivirus (MDA), on the other hand, is a modern EPP enabled by default on any fresh Windows installation. MDA is the focus of this analysis.

Because of its widespread adoption, MDA has been an interesting target of security researchers for quite a while. Given its size and complexity though, each analysis tends to focus on a specific component. For instance, some research targeted the emulator [WindowsOffender], others the minifilter driver [WdFilter1], and others the ELAM driver [WdBoot]. Further research was focused on the signature file format [WdExtract], while one of the most recent studies targeted the signature update system [Pretender].

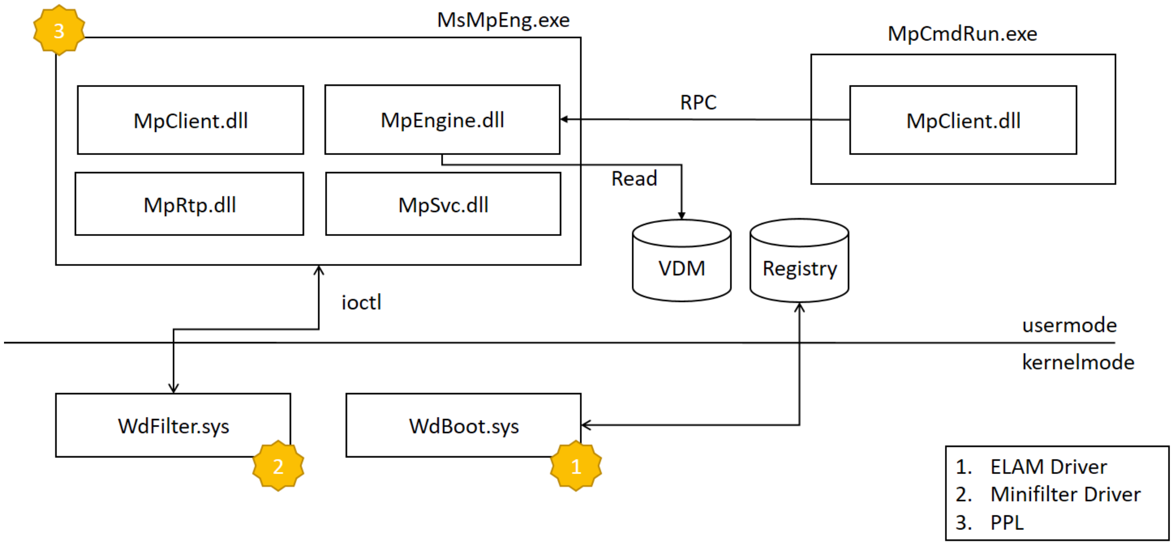
In this blog post, we will continue working on the MDA signatures, more specifically we focus on:

- The signature database
- The loading process of the signatures
- The types and layout of different signatures
- A detailed discussion on two signature types: `PEHSTR` and `PEHSTR_EXT`

The goal of RETooling is to provide the best tools for offensive teams. Adversary emulation is nowadays becoming an important practice for many organizations. In this context, understanding the inner workings of security products is crucial to replicate threat actors activities safely and reliably.

1-Jul-2024 Update : We gave a workshop on MDA signatures at Recon 2024. You can access the workshop materials at our GitHub repository: workshop-recon24.

Microsoft Defender Antivirus Architecture



The MDA product is composed of modules running both in kernel and user mode. The overview is depicted in Figure 1. The first component which is loaded is the `WdBoot.sys`. This is the ELAM driver that checks the integrity of the system during the early stages of the system boot. It is loaded **before** any other third party driver and it scans each loaded driver image **before** its `DriverEntry` is invoked. For the detection it uses its own set of signatures that are stored on a special registry Value Key (`HKLM\ELAM\Microsoft Antimalware Platform\Measured`) which is not accessible after the ELAM driver is unloaded.

The *Microsoft Defender Antivirus Service* main responsibility is to start the main MDA executable, namely `MsMpEng.exe` . Such process is executed with `EPROCESS.Protection` equal to `AntimalwareLight` (`0x31`) thanks to the `WdBoot` certification. `MsMpEng` is a relatively small (~300K) executable which loads the following bigger components that implement most of the logic:

- `MsRtp`: it manages the Real-time protection
- `MpSvc`: it loads and manages the main component `MpEngine`
- `MpEngine`: is the biggest component (~19 MB). It implements scanners, emulators, modules, signature loading from the *VDM file* and signature handling.

`MpCmdRun` is an external command line tool that uses the `MpClient` library to interact with the main service. `MpClient` is an auxiliary library that implements a bunch of RPC requests for the service (to get the configuration or to request a scan). Last but not least, there is the `WdFilter.sys`, the main kernel space component of the MDA architecture. It monitors the access to the filesystem by registering as minifilter driver, it registers notification routines (image load, process creation, object access etc.) and more.

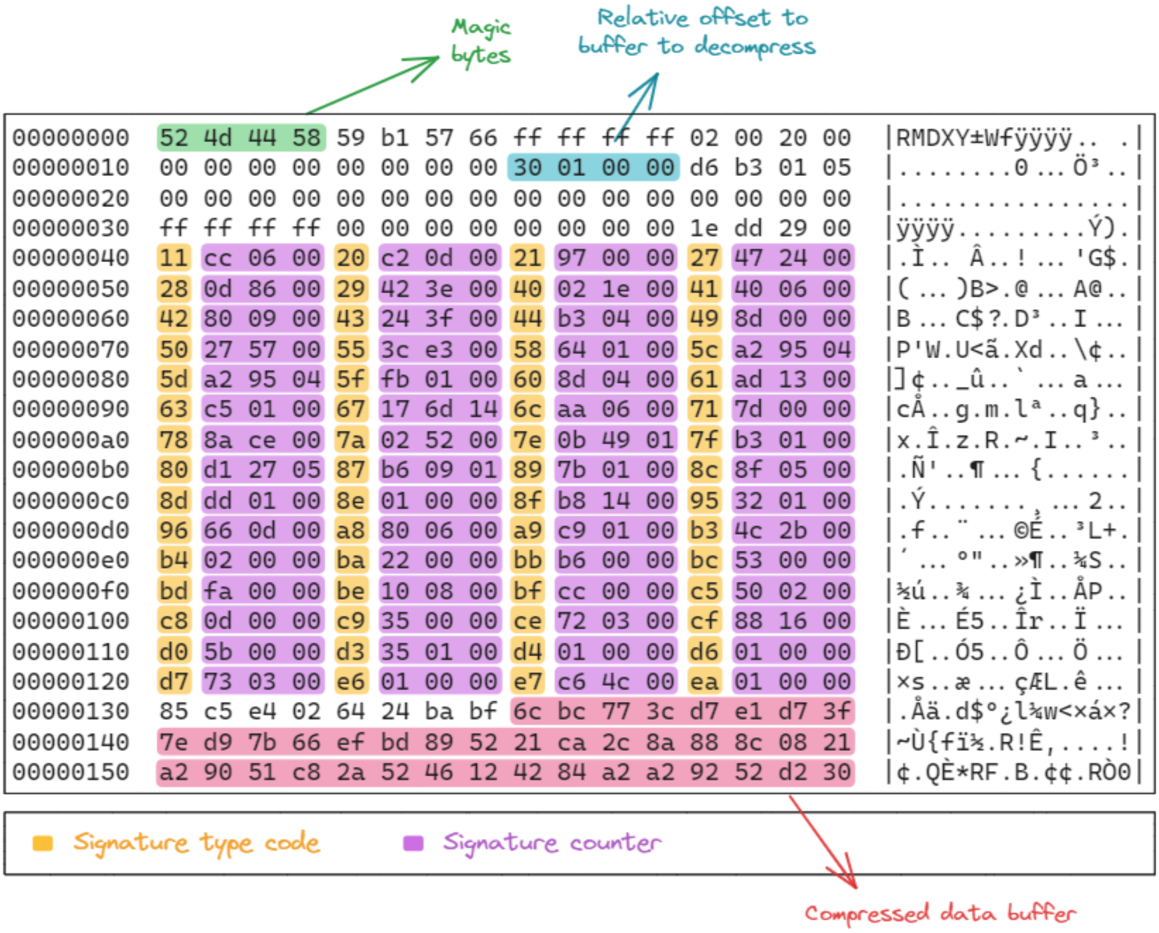
The analysis was performed on the product version `1.1.23100` (October 2023 release) and the signature version `1.401.1166.0`.

The signature database

The MDA signatures are distributed in four different `.vdm` files:

- `mpavbase.vdm`: released with platform updates (typically once per month), contains the anti-malware signatures
- `mpasbase.vdm`: released with platform updates (typically once per month), contains the anti-spyware signatures
- `mpavdlta.vdm`: released on a daily basis, contains only the new signatures that are merged in memory with `mpavbase.vdm` when the database is loaded at runtime
- `mpasdlta.vdm`: released on a daily basis, contains only the new signatures that are merged in memory with `mpasbase.vdm` when the database is loaded at runtime

Those files are located in `C:\ProgramData\Microsoft\Windows Defender\Definition Updates\<RandomGUID>\` .



```
00000000  52 4d 44 58 59 b1 57 66 ff ff ff ff 02 00 20 00  |RMDXY±Wfÿÿÿÿ.. .|
00000010  00 00 00 00 00 00 00 00 30 01 00 00 d6 b3 01 05  |........0...Ö³..|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000030  ff ff ff ff 00 00 00 00 00 00 00 00 1e dd 29 00  |ÿÿÿÿ.........Ý).|
00000040  11 cc 06 00 20 c2 0d 00 21 97 00 00 27 47 24 00  |.Ì.. Â..!...'G$.|
00000050  28 0d 86 00 29 42 3e 00 40 02 1e 00 41 40 06 00  |(...)B>.@...A@..|
00000060  42 80 09 00 43 24 3f 00 44 b3 04 00 49 8d 00 00  |B...C$?.D³..I...|
00000070  50 27 57 00 55 3c e3 00 58 64 01 00 5c a2 95 04  |P'W.U<ã.Xd..\¢..|
00000080  5d a2 95 04 5f fb 01 00 60 8d 04 00 61 ad 13 00  |]¢.._û..`...a...|
00000090  63 c5 01 00 67 17 6d 14 6c aa 06 00 71 7d 00 00  |cÅ..g.m.lª..q}..|
000000a0  78 8a ce 00 7a 02 52 00 7e 0b 49 01 7f b3 01 00  |x.Î.z.R.~.I..³..|
000000b0  80 d1 27 05 87 b6 09 01 89 7b 01 00 8c 8f 05 00  |.Ñ'..¶...{......|
000000c0  8d dd 01 00 8e 01 00 00 8f b8 14 00 95 32 01 00  |.Ý.......¸...2..|
000000d0  96 66 0d 00 a8 80 06 00 a9 c9 01 00 b3 4c 2b 00  |.f..¨...©É..³L+.|
000000e0  b4 02 00 00 ba 22 00 00 bb b6 00 00 bc 53 00 00  |´...º"..»¶..¼S..|
000000f0  bd fa 00 00 be 10 08 00 bf cc 00 00 c5 50 02 00  |½ú..¾...¿Ì..ÅP..|
00000100  c8 0d 00 00 c9 35 00 00 ce 72 03 00 cf 88 16 00  |È...É5..Îr..Ï...|
00000110  d0 5b 00 00 d3 35 01 00 d4 01 00 00 d6 01 00 00  |Ð[..Ó5..Ô...Ö...|
00000120  d7 73 03 00 e6 01 00 00 e7 c6 4c 00 ea 01 00 00  |×s..æ...çÆL.ê...|
00000130  85 c5 e4 02 64 24 ba bf 6c bc 77 3c d7 e1 d7 3f  |.Åä.d$º¿l¼w<×á×?|
00000140  7e d9 7b 66 ef bd 89 52 21 ca 2c 8a 88 8c 08 21  |~Ù{fï½.R!Ê,....!|
00000150  a2 90 51 c8 2a 52 46 12 42 84 a2 a2 92 52 d2 30  |¢.QÈ*RF.B.¢¢.RÒ0|
```

Both `mpavbase.vdm` and `mpasbase.vdm` are Portable Executable files, containing a resource with the compressed signatures inside. Such resource has a global header which starts with a magic value equal to `RMDX`.

The Figure 2 represents the global header. The `uint32` field at offset `0x18` from the beginning of the resource contains the relative offset (`0x130` in figure) which points to the the payload. Indeed, after 8 bytes from this point, the compressed data buffer starts.

At offset `0x40` the global header has an array of `DWORD`. Each `DWORD` is split in two parts: the most significant byte (in yellow) represents the signature type whereas the remaining three bytes (in purple) represent the number of signature of that type in reverse byte order.

The payload starting at offset `0x138` is compressed with the gzip algorithm and can be extracted with the following python code.

```
import zlib

compressed = open('x.gz', 'rb').read()
decompressed = zlib.decompress(compressed, -zlib.MAX_WBITS)
```

Start your MpEngine

The core logic of the MDA is located into the `MpEngine.dll`. The engine is loaded by the `MpSvc.dll` in the `MpSvc!LoadEngine` function which invokes the `MpSvc!InitEngineContext`. Here the `MpEngine.dll` is loaded though `KernelBase!LoadLibraryExW` and the address of one of the core functions that allows the service to complete the initialization of the engine is retrieved: the export `mpengine!__rsignal`. Roughly speaking, the `rsignal` function is essentially a wrapper of the function `mpengine!DispatchSignalOnHandle` which calls the function corresponding to the input parameter `signal_id`.

Here is the prototype of the function:

```
UINT64 DispatchSignalOnHandle(
                PVOID g_hSignalptr,
                UINT64 signal_id,
                BYTE *pParams,
                SIZE_T cbParams
)
```

To trigger the initialization of the `MpEngine`, the `MpSvc!InitEngineContext` invokes the `rsignal` function as follows:

```
(pMpengineCtx->pfn__rsignal)(
                    &pMpengineCtx->hMpEngineInstance,
                    0x4036,
                    pParam,
                    0x1B8
);
```

Where the parameters are described below:

- `pMpengineCtx->hMpEngineInstance`: is an output parameter that receives the handle to the initialized engine
- the `0x4036` is the `signal_id` for an initialization request
- the `pParams` points to the initialization parameters
- the `cbParams` is the size of `pParams` in bytes

The MpEngine function that corresponds to the signal 0x4036 is the StartMpEngine. This function recursively calls DispatchSignalOnHandle with the following relevant signal ids:

- 0x4019: triggers the execution of the InitializeMpEngine which is in charge of initializing a core data structure named gktab. The gktab is huge (0x15bb0 bytes!) and contains tens of fields, however the first (and probably one of the most relevant ones) is the pointer to the function ksignal which implements the majority of signal_id-related functionalities (more than **50**)
- 0x401a: invokes the ksignal and dispatch function to manage defender exclusions
- 0x400b: invokes the ksignal function to call the modprobe_init function where the modules' initialization happens

MpEngine Modules Initialization

The MpEngine contains a lot of modules (named AutoInitModules). Some of them are used to introduce the support for specific file formats such as PE, ELF, Mach-O and implement specialized scanners (that are registered during the initialization through the ScanRegister function) and others implement auxiliary functionalities, such as signature loaders.

Such modules are referenced by the global array g_pUniModEntries. Each entry is a struct of type unimod_entry_t :

```
struct unimod_entry_t
{
    PCHAR pModuleName;
    PVOID pfnInit;
    PVOID pfnCleanup;
    __int64 Unk;
};
```

Where pModuleName is a human readable name of the module, the pfnInit is a pointer to the initialization function and pfnCleanup points to the cleanup function. The last field is typically 0 or 1 but we did not care much about it.

As described earlier the MpSvc!InitEngineContext calls the __rsignal function with several signal_id and one of those is the one that triggers the execution of the modprobe_init function which will be described in the remaining part of this section.

The modprobe_init function operates in three main phases:

1. **Preloading of the Signature Databases**
    - This phase involves parsing the main header and initializing necessary data structures.
    - This is achieved by calling the mpengine!preload_database function.

2. **Execution of the Initialization Functions**

    - The initialization functions of all the `AutoInitModules` are called.
    - This is done by looping over all the entries in the `unimod_entry_t` of the `g_pUniModEntries` and calling the `pfnInit` function for each entry.

3. **Finalization of the Signature Loading Process**

    - The final phase involves completing the signature loading process.
    - This is done by calling the `mpengine!load_database` function.

Making a comprehensive description of all the modules would require an enormous amount of work. As such, for our purposes we will focus only on two of them: the `cksig` and the `pefile`.

The `cksig` Module

The `cksig` module falls in the category of *signature loaders* and it is initialized within the `mpengine!cksig_init_module`. In this phase of the init process, the database has not been loaded yet. Since the semantics of each signature is different from the other, the engine has a dedicated *loader* for each type of signature. A designated function reads the body of the signature and loads it into memory. A module that wants to handle a specific signature format must register a callback with the engine.

To support this process, if a module wants to handle a specific signature format it must register a callback with the engine. Later, when the signature loading process is finalized (Phase 3), the callback will be invoked for each record loaded from the VDM file.

As far as we know there are two functions to register a callback: `mpengine!RegisterForDatabaseRecords` and `mpengine!regcntl`. The `mpengine!RegisterForDatabaseRecords` function takes in input the address of a global variable that receives the handle to a signature type and the loader callback.

The `regcntl` takes in input a `hstr_handler` object defined as follows:

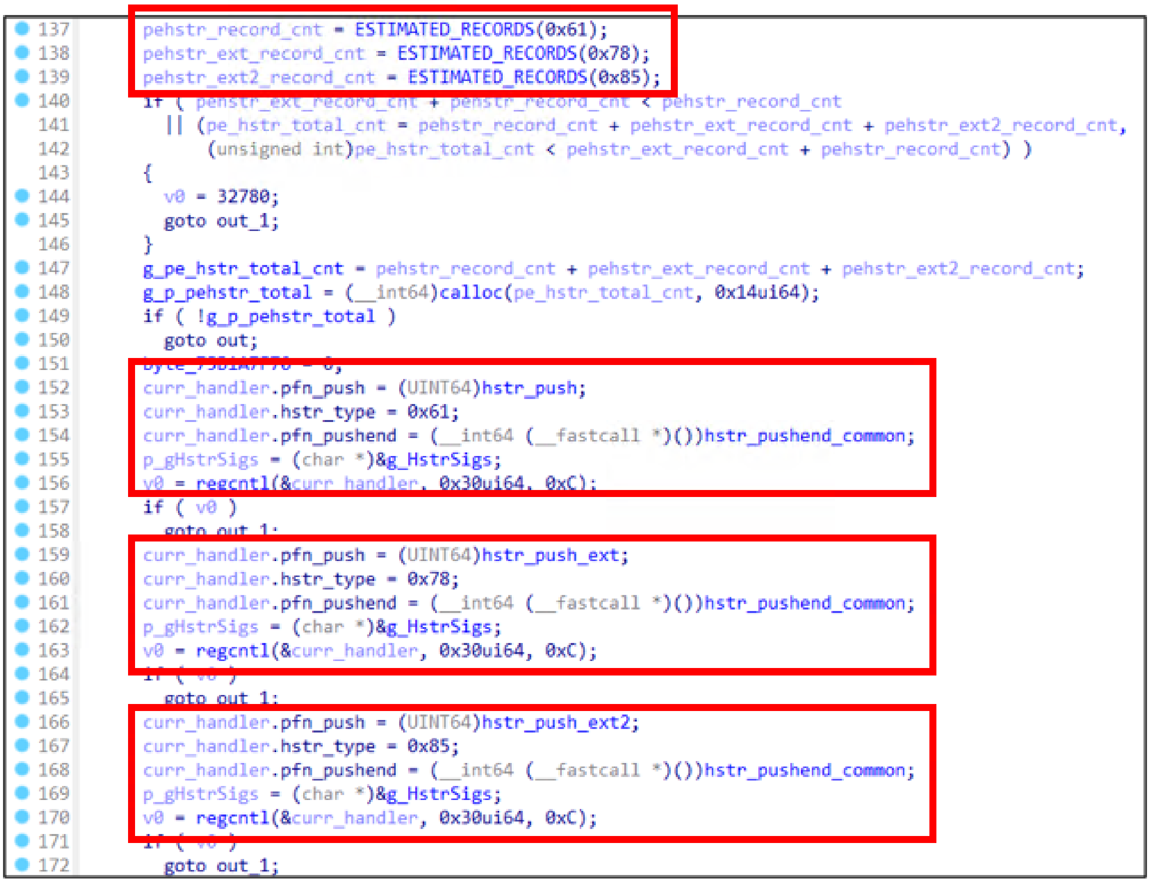
```
struct __declspec(align(8)) hstr_handler
{
    UINT64 (__stdcall *pfn_push)(UINT64, UINT16 *, UINT64, UINT64, UINT32);
    UINT64 pHstrSigs;
    UINT8 hstr_type;
    UINT64 (__stdcall *pfn_pushend)(UINT64);
    UINT64 (__stdcall *pfn_unload)(UINT64);
    PVOID pHstrSigs2;
};
```

where the `pfn_push` is the function pointer to the handler of signature of type `hstr_type`. The `pHstrSigs` and the `pHstrSigs2` point to signature records for the current `hstr_type`. The `pfn_pushend` and `pfn_unload` are other two functions part of the signature handling that are not covered by this post.

The cksig module uses both RegisterForDatabaseRecords and regcntl to register two different groups of signatures. We will focus on the latter because it targets the HSTR signatures. The initialization of the second group is done within the pattsearch_init subroutine (invoked by cksig_init_module) which set the callbacks for a specific **signature family** named HSTR. Such family includes the following signature types:

ID	SIGNATURE TYPE
97	SIGNATURE_TYPE_PEHSTR
120	SIGNATURE_TYPE_PEHSTR_EXT
133	SIGNATURE_TYPE_PEHSTR_EXT2
140	SIGNATURE_TYPE_ELFHSTR_EXT
141	SIGNATURE_TYPE_MACHOHSTR_EXT
142	SIGNATURE_TYPE_DOSHSTR_EXT
143	SIGNATURE_TYPE_MACROHSTR_EXT
190	SIGNATURE_TYPE_DEXHSTR_EXT
191	SIGNATURE_TYPE_JAVAHSTR_EXT
197	SIGNATURE_TYPE_ARHSTR_EXT
209	SIGNATURE_TYPE_SWFHSTR_EXT
211	SIGNATURE_TYPE_AUTOITHSTR_EXT
212	SIGNATURE_TYPE_INNOHSTR_EXT
215	SIGNATURE_TYPE_CMDHSTR_EXT
228	SIGNATURE_TYPE_MDBHSTR_EXT
234	SIGNATURE_TYPE_DMGHSTR_EXT

In order to set this new group of handlers for the HSTR signature family, it first computes the number of records and allocates a contiguous memory area of 0x14 * hstr_total_cnt (line 148 of the disassembled code below shows the example for the PEHSTR sub-family).



```
pehstr_record_cnt = ESTIMATED_RECORDS(0x61);
pehstr_ext_record_cnt = ESTIMATED_RECORDS(0x78);
pehstr_ext2_record_cnt = ESTIMATED_RECORDS(0x85);
if ( pehstr_ext_record_cnt + pehstr_record_cnt < pehstr_record_cnt
  || (pe_hstr_total_cnt = pehstr_record_cnt + pehstr_ext_record_cnt + pehstr_ext2_record_cnt,
      (unsigned int)pe_hstr_total_cnt < pehstr_ext_record_cnt + pehstr_record_cnt) )
{
  v0 = 32780;
  goto out_1;
}
g_pe_hstr_total_cnt = pehstr_record_cnt + pehstr_ext_record_cnt + pehstr_ext2_record_cnt;
g_p_pehstr_total = (__int64)calloc(pe_hstr_total_cnt, 0x14ui64);
if ( !g_p_pehstr_total )
  goto out;
byte_75B1A7770 = 0;
curr_handler.pfn_push = (UINT64)hstr_push;
curr_handler.hstr_type = 0x61;
curr_handler.pfn_pushend = (__int64 (__fastcall *)())hstr_pushend_common;
p_gHstrSigs = (char *)&g_HstrSigs;
v0 = regcntl(&curr_handler, 0x30ui64, 0xC);
if ( v0 )
  goto out_1;
curr_handler.pfn_push = (UINT64)hstr_push_ext;
curr_handler.hstr_type = 0x78;
curr_handler.pfn_pushend = (__int64 (__fastcall *)())hstr_pushend_common;
p_gHstrSigs = (char *)&g_HstrSigs;
v0 = regcntl(&curr_handler, 0x30ui64, 0xC);
if ( v0 )
  goto out_1;
curr_handler.pfn_push = (UINT64)hstr_push_ext2;
curr_handler.hstr_type = 0x85;
curr_handler.pfn_pushend = (__int64 (__fastcall *)())hstr_pushend_common;
p_gHstrSigs = (char *)&g_HstrSigs;
v0 = regcntl(&curr_handler, 0x30ui64, 0xC);
if ( v0 )
  goto out_1;
```

The `ESTIMATED_RECORDS` function takes in input the signature type and returns the number of signature present in the VDM for that signature type (namely signature `0x61` , `0x78` and `0x85` in figure). Remember that the information on the number of records per signature type is stored into the global header of the VDM (before the compressed data), so it is available after the *preloading* phase.

The example in Figure 3 uses the `regcntl` function to register the callback. The handler for the specific hstr type is passed in input (`hstr_push`, `hstr_push_ext`, `hstr_push_ext2` in figure)

Notably, all the `PEHSTR` point to the same `hstr.pHstrSigs`. This is not true for other types of `HSTR` signatures. We also found references to dynamic `HSTR` signatures that are probably related to `Microsoft Active Protection Service (MAPS)` (aka cloud delivered protection) but we did not investigate that part.

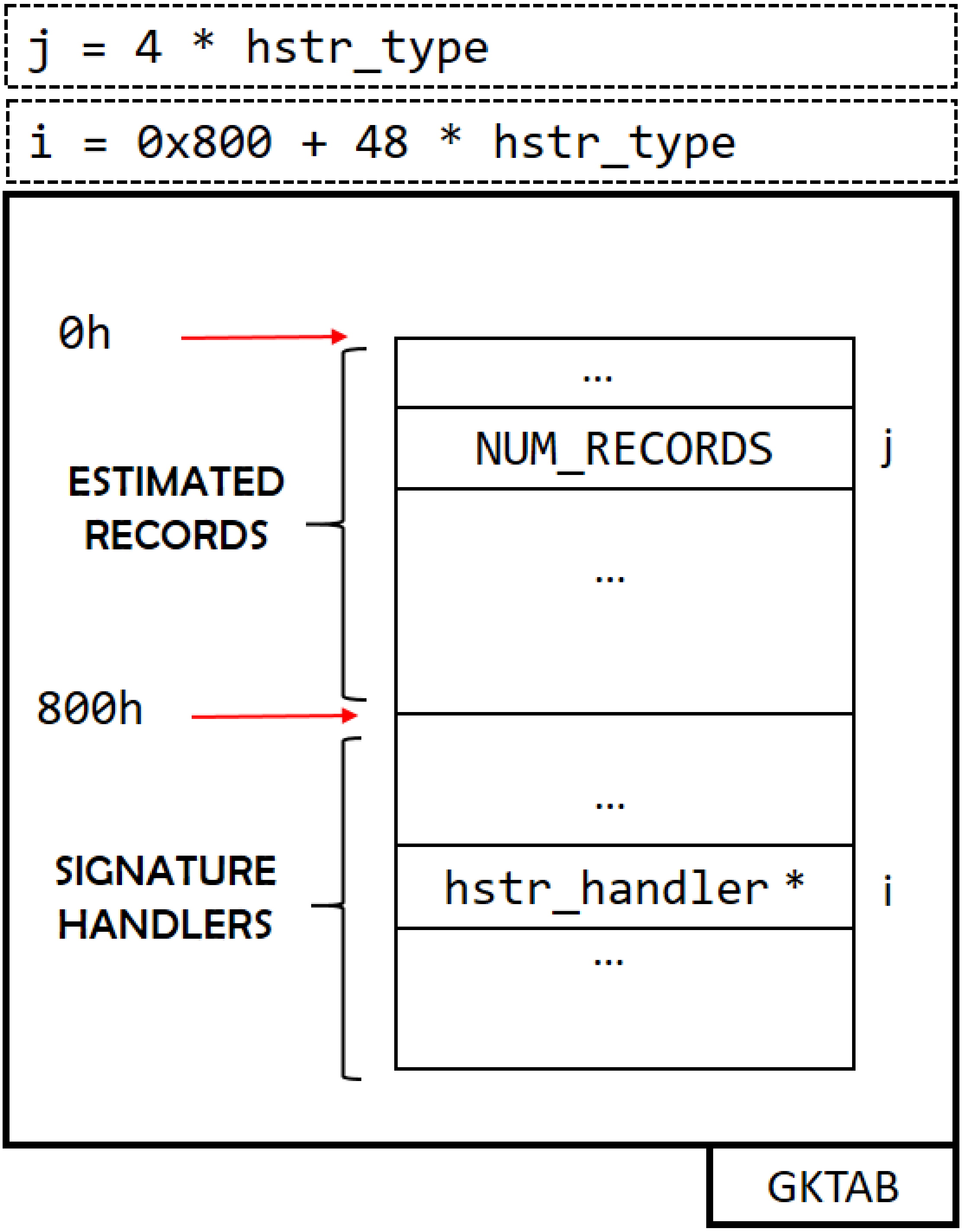


Figure 4 represents how the record numbers and the signature handlers are stored in memory within the `gktab` data structure. The first part is an array of `DWORD` each containing the number of records for a given signature and, starting from offset `0x800`, we can observe a second array containing pointers to `struct hstr_handler`.

Completing the signature loading process

So far we talked about the *preloading* and the *initialization* phases of the `modprobe_init` function. In order to make the MDA signature subsystem up and running one last step must be completed. This is done in the `mpengine!load_database` function which invokes the mpengine!DispatchRecords function. Here, each entry of the VDM is processed and, based on the signature type, the signature payload is *dispatched* to the handler for that signature type (registered by the CKSIG module in the previous step).

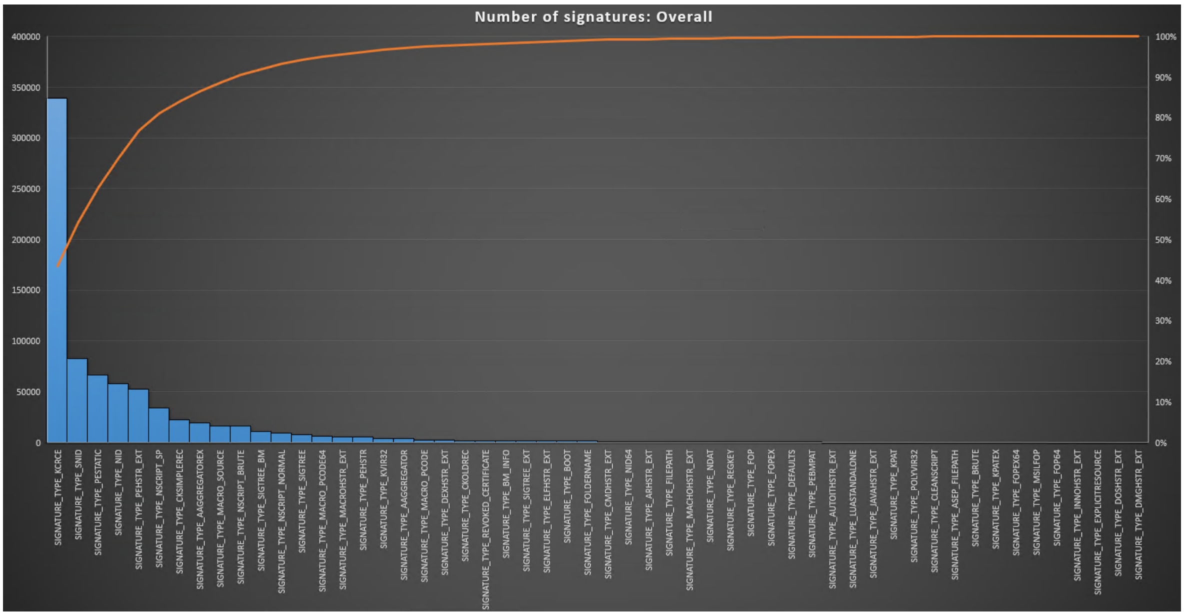
pefile_module and the hstr_internal_search

This pefile_module module belongs to the **scanners** type. Indeed one of the first action invoked during its initialization is to call mpengine!ScanRegister. The first parameter is a function pointer which implements the actual scan (pefile_scan for the pefile module).

Whenever a PE file scan is triggered, in order to match the signature with the HSTR signatures, the hstr_internal_search is invoked. We noticed two different call stacks: one that goes through the mpengine!scan_vbuff and the other that goes through mpengine!scan_vmem. This is probably due to the fact that the scan can be triggered multiple times during the emulation. The hstr_internal_search finds the right file type offset base within the g_HstrSigs array (which is 0 for PE files) and than calls the hstr_internal_search_worker that implements the actual search.

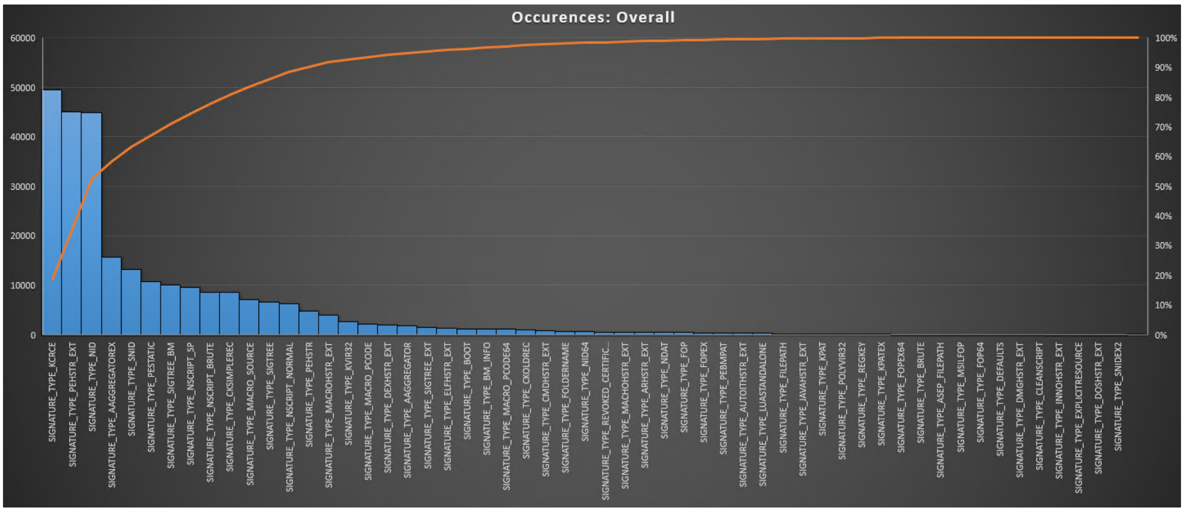
Signature stats

Before delving into the signature type analysis we report few stats on the different signature types that are present into the VDM file. To have an idea of signature distribution we wrote a simple python script that parses the VDM and counts the number of signatures by types.

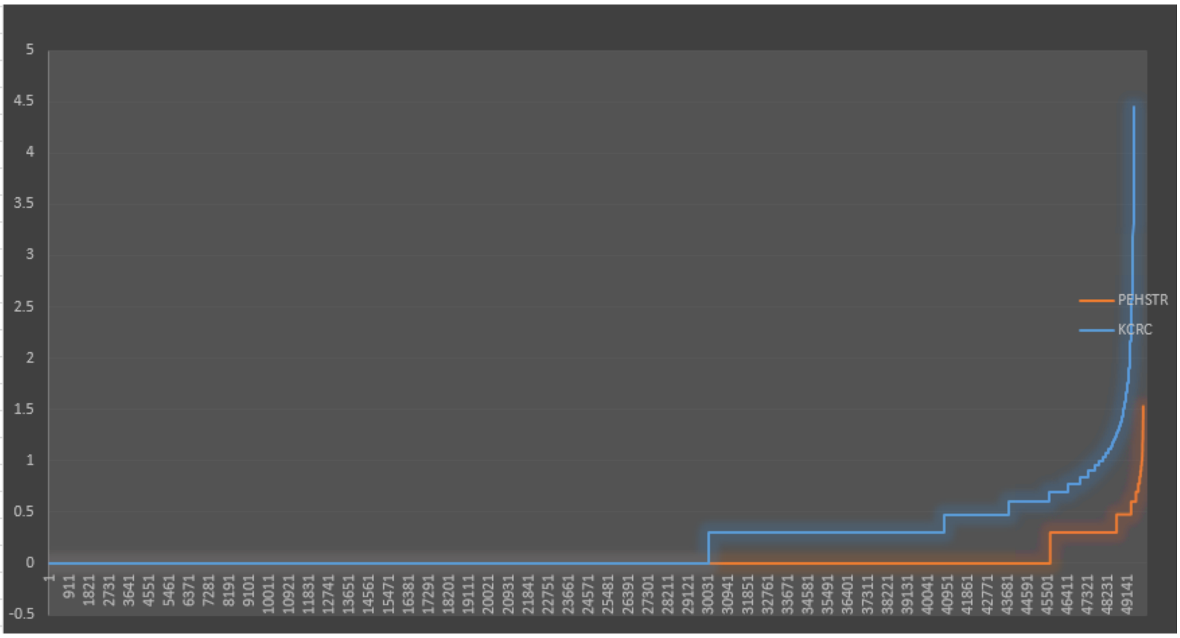


We noticed that the SIGNATURE_TYPE_KCRCE is by far the most frequent with more than 338k signatures. Indeed, even by stacking all the HSTR family signature types together we don't even reach 100k signatures. Far below the KCRCE.

However, by looking at the number of *occurrences* per threat, things changes significantly. By *occurrences* we mean that we set a bit to one if a given signature type appears on a given threat and 0 otherwise. In other words, we do not consider the actual number of signature but we look at their distribution.



As the histogram shows, the `PEHSTR_EXT` moved very close to the `KCRCE`. If we sum the `PEHSTR_*` signatures together, we get a greater value than the `KCRCE` alone. Indeed if we look at the distribution of the `KCRCE` signature we notice that the **20%** of the threats account for a very high percentage of the `KCRCE` signatures. Whereas the `PEHSTR` are much more evenly distributed.



Starting from the analysis of the signature distribution we decided to focus on the `HSTR`, and particularly on the `PEHSTR` .

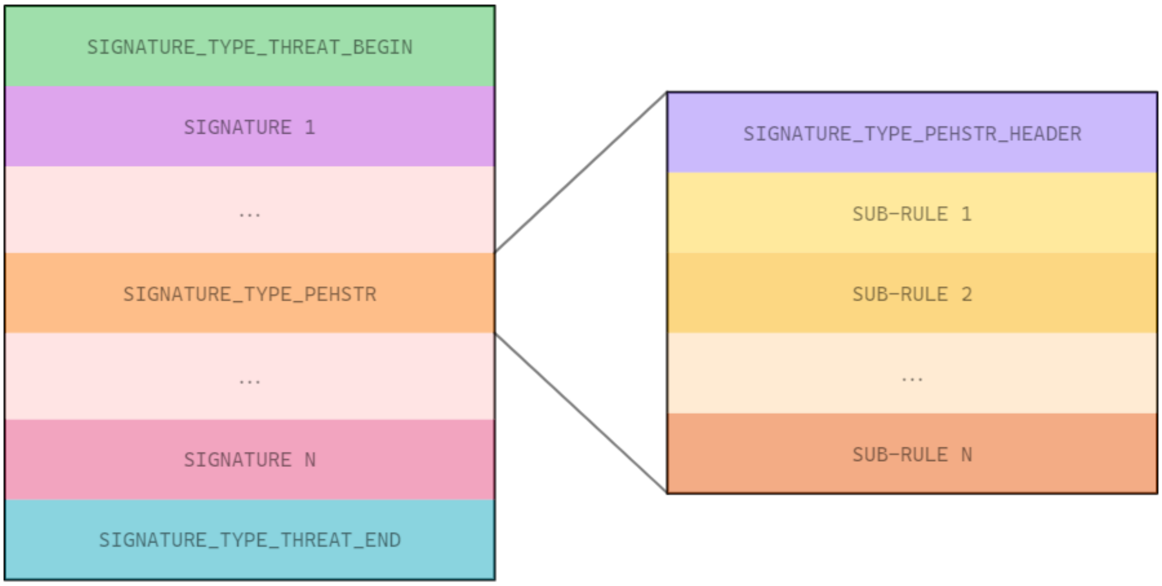
Let's take a closer look at them!

Signature main struct

The signatures contained in mpavbase.vdm and mpasbase.vdm follow a hierarchical composition structure. This structure organizes the threat signatures in a specific manner. Each threat, representing a particular type of malware, is defined by a set of one or more signatures. These signatures are used to identify and detect the presence of the corresponding threat.

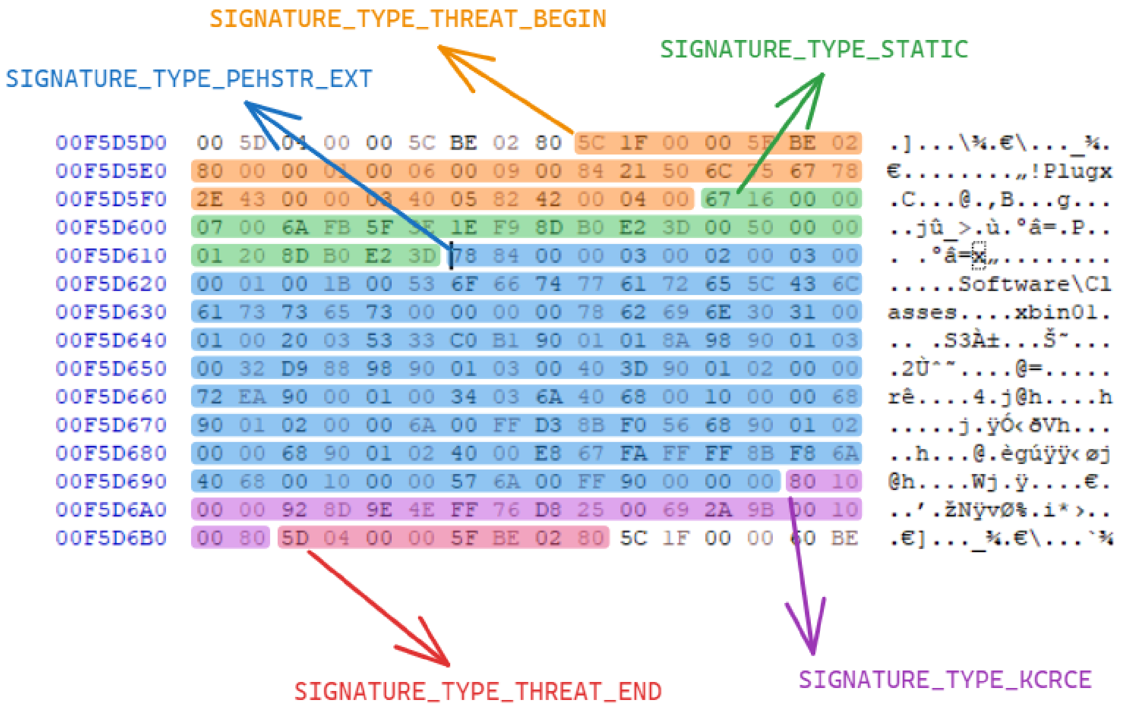
The organization of threats and their associated signatures within the files is delineated, in turn, by a specific types of signatures.. The beginning of a threat's signature set is marked by SIGNATURE_TYPE_THREAT_BEGIN, while the end of the set is marked by SIGNATURE_TYPE_THREAT_END. These delimiters enclose the collection of signatures that collectively define and identify a particular threat.

The conceptual structure of how threats and their signatures are organized within these files is depicted in **Figure 5**. The figure provides a visual representation of the hierarchical composition, illustrating how threats are defined by multiple signatures and how these signatures are grouped together using the designated begin and end markers.



For example, **Figure 6** shows an entire threat actor with name Plugx.C, containing various signatures used to detect it. In between the SIGNATURE_TYPE_THREAT_BEGIN and SIGNATURE_TYPE_THREAT_END there are three different signatures, in order:

- SIGNATURE_TYPE_STATIC: highlighted in green;
- SIGNATURE_TYPE_PEHSTR_EXT: highlighted in blue;
- SIGNATURE_TYPE_KCRCE: highlighted in violet;


SIGNATURE_TYPE_THREAT_BEGIN
SIGNATURE_TYPE_STATIC
SIGNATURE_TYPE_PEHSTR_EXT
SIGNATURE_TYPE_THREAT_END
SIGNATURE_TYPE_KCRCE

```
00F5D5D0  00 5D 04 00 00 5C BE 02 80 5C 1F 00 00 5F BE 02  .]...\¾.€\..._¾.
00F5D5E0  80 00 00 01 00 06 00 09 00 84 21 50 6C 75 67 78  €........„!Plugx
00F5D5F0  2E 43 00 00 03 40 05 82 42 00 04 00 67 16 00 00  .C...@.‚B...g...
00F5D600  07 00 6A FB 5F 3E 1E F9 8D B0 E2 3D 00 50 00 00  ..jû_>.ù.°â=.P..
00F5D610  01 20 8D B0 E2 3D 78 84 00 00 03 00 02 00 03 00  . .°â=x„........
00F5D620  00 01 00 1B 00 53 6F 66 74 77 61 72 65 5C 43 6C  .....Software\Cl
00F5D630  61 73 73 65 73 00 00 00 00 78 62 69 6E 30 31 00  asses....xbin01.
00F5D640  01 00 20 03 53 33 C0 B1 90 01 01 8A 98 90 01 03  .. .S3À±...Š˜...
00F5D650  00 32 D9 88 98 90 01 03 00 40 3D 90 01 02 00 00  .2Ùˆ˜....@=.....
00F5D660  72 EA 90 00 01 00 34 03 6A 40 68 00 10 00 00 68  rê....4.j@h....h
00F5D670  90 01 02 00 00 6A 00 FF D3 8B F0 56 68 90 01 02  .....j.ÿÓ‹ðVh...
00F5D680  00 00 68 90 01 02 40 00 E8 67 FA FF FF 8B F8 6A  ..h...@.ègúÿÿ‹øj
00F5D690  40 68 00 10 00 00 57 6A 00 FF 90 00 00 00 80 10  @h....Wj.ÿ....€.
00F5D6A0  00 00 92 8D 9E 4E FF 76 D8 25 00 69 2A 9B 00 10  ..’.žNÿvØ%.i*›..
00F5D6B0  00 80 5D 04 00 00 5F BE 02 80 5C 1F 00 00 60 BE  .€]..._¾.€\...`¾
```

By employing this hierarchical composition approach, Microsoft Defender Antivirus can effectively manage and maintain a comprehensive database of threat signatures. This allows for efficient detection and protection against a broad spectrum of security threats.

Windows Defender signatures follow a common structure defined as:

```
typedef struct _STRUCT_COMMON_SIGNATURE_TYPE {
    UINT8  ui8SignatureType;
    UINT8  ui8SizeLow;
    UINT16 ui16SizeHigh;
    BYTE   pbRuleContent[];
} STRUCT_COMMON_SIGNATURE_TYPE, *PSTRUCT_COMMON_SIGNATURE_TYPE;
```

In this structure:

- `ui8SignatureType` specifies the type of the signature.
- `ui8SizeLow` indicates the lower byte size of the signature.
- `ui16SizeHigh` represents the higher byte size of the signature.
- `pbRuleContent[]` contains the rule content, with the total size calculated as: `ui8SizeLow | (ui16SizeHigh << 8)`.

SIGNATURE_TYPE_THREAT_BEGIN and THREAT_END

A threat actor is represented in this context as a sequence of different types of signatures used to detect it. These signatures are contained between the two aforementioned types.

The signatures `SIGNATURE_TYPE_THREAT_BEGIN` and `SIGNATURE_TYPE_THREAT_END` are not simple markers but contain different information.

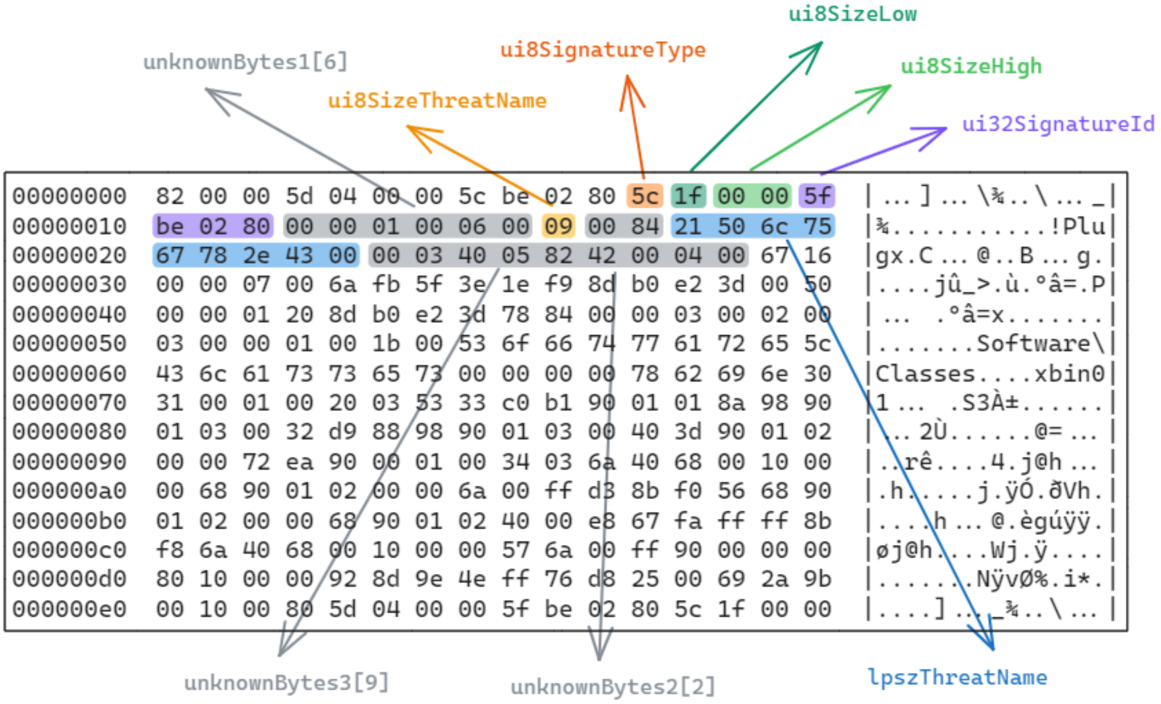
SIGNATURE_TYPE_THREAT_BEGIN has the following structure:

```
typedef struct _STRUCT_SIG_TYPE_THREAT_BEGIN {
    UINT8  ui8SignatureType;
    UINT8  ui8SizeLow;
    UINT16 ui16SizeHigh;
    UINT32 ui32SignatureId;
    BYTE   unknownBytes1[6];
    UINT8  ui8SizeThreatName;
    BYTE   unknownBytes2[2];
    CHAR   lpszThreatName[ui8SizeThreatName];
    BYTE   unknownBytes3[9];
} STRUCT_SIG_TYPE_THREAT_BEGIN,* PSTRUCT_SIG_TYPE_THREAT_BEGIN;
```

where:

- ui8SignatureType: a hexadecimal code defining the type of signature (0x05C).
- ui8SizeLow: the low part of the size of the entire signature.
- ui16SizeHigh: the high part of the size of the entire signature.
- ui32SignatureId: the identifier of the signature, used by mpengine.dll.
- unknownBytes1: six unknown bytes.
- ui8SizeThreatName: represents the size in bytes of the threat name.
- unknownBytes2: two unknown bytes.
- lpszThreatName: a string representing the threat name.
- unknownBytes3: nine unknown bytes.

An example of the SIGNATURE_TYPE_THREAT_BEGIN is shown in **Figure 7**:

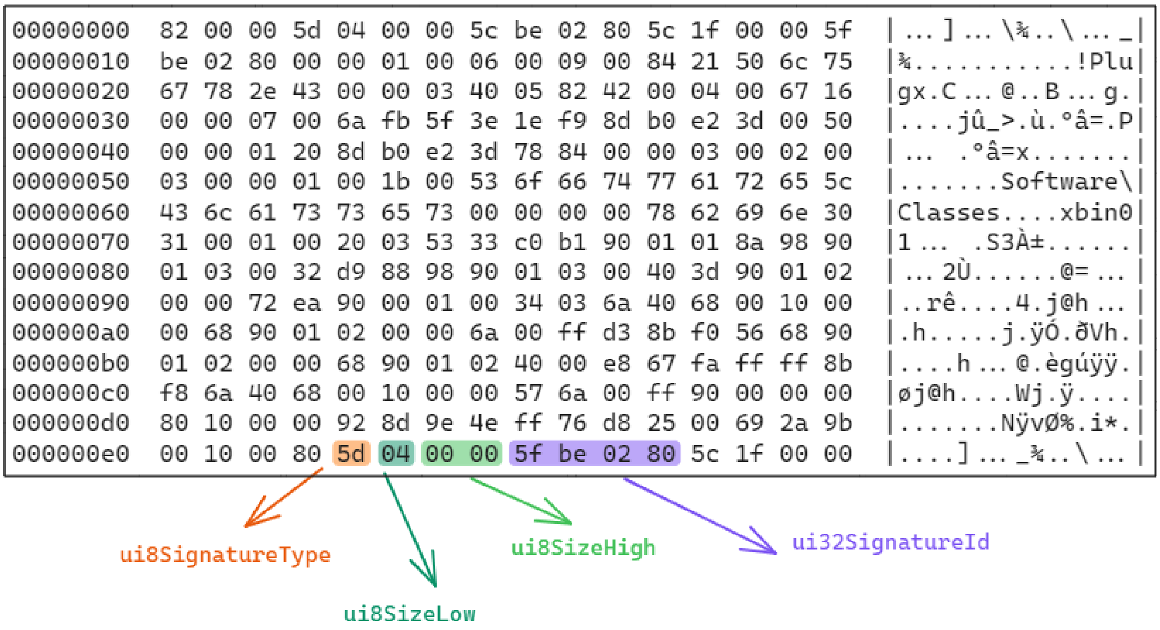


```
00000000  82 00 00 5d 04 00 00 5c be 02 80 5c 1f 00 00 5f  |...]...\¾..\..._|
00000010  be 02 80 00 00 01 00 06 00 09 00 84 21 50 6c 75  |¾...........!Plu|
00000020  67 78 2e 43 00 00 03 40 05 82 42 00 04 00 67 16  |gx.C...@..B...g.|
00000030  00 00 07 00 6a fb 5f 3e 1e f9 8d b0 e2 3d 00 50  |....jû_>.ù.°â=.P|
00000040  00 00 01 20 8d b0 e2 3d 78 84 00 00 03 00 02 00  |... .°â=x.......|
00000050  03 00 00 01 00 1b 00 53 6f 66 74 77 61 72 65 5c  |.......Software\|
00000060  43 6c 61 73 73 65 73 00 00 00 00 78 62 69 6e 30  |Classes....xbin0|
00000070  31 00 01 00 20 03 53 33 c0 b1 90 01 01 8a 98 90  |1... .S3À±......|
00000080  01 03 00 32 d9 88 98 90 01 03 00 40 3d 90 01 02  |...2Ù......@=...|
00000090  00 00 72 ea 90 00 01 00 34 03 6a 40 68 00 10 00  |..rê....4.j@h...|
000000a0  00 68 90 01 02 00 00 6a 00 ff d3 8b f0 56 68 90  |.h.....j.ÿÓ.ðVh.|
000000b0  01 02 00 00 68 90 01 02 40 00 e8 67 fa ff ff 8b  |....h...@.ègúÿÿ.|
000000c0  f8 6a 40 68 00 10 00 00 57 6a 00 ff 90 00 00 00  |øj@h....Wj.ÿ....|
000000d0  80 10 00 00 92 8d 9e 4e ff 76 d8 25 00 69 2a 9b  |.......NÿvØ%.i*.|
000000e0  00 10 00 80 5d 04 00 00 5f be 02 80 5c 1f 00 00  |....]..._¾..\...|
```

STRUCT_SIG_TYPE_THREAT_END has the generic signature format:

```
typedef struct _STRUCT_SIG_TYPE_THREAT_END {
  UINT8  ui8SignatureType;
  UINT8  ui8SizeLow;
  UINT16 ui16SizeHigh;
  BYTE   pbRuleContent[];
} STRUCT_SIG_TYPE_THREAT_END,* PSTRUCT_SIG_TYPE_THREAT_END;
```

where `ui8SignatureType` has the value `0x5D`, and the `pbRuleContent` value is the same as the corresponding `ui32SignatureId` used in `SIGNATURE_TYPE_THREAT_BEGIN`.



```
00000000  82 00 00 5d 04 00 00 5c be 02 80 5c 1f 00 00 5f  |...]...\¾..\..._|
00000010  be 02 80 00 00 01 00 06 00 09 00 84 21 50 6c 75  |¾...........!Plu|
00000020  67 78 2e 43 00 00 03 40 05 82 42 00 04 00 67 16  |gx.C...@..B...g.|
00000030  00 00 07 00 6a fb 5f 3e 1e f9 8d b0 e2 3d 00 50  |....jû_>.ù.°â=.P|
00000040  00 00 01 20 8d b0 e2 3d 78 84 00 00 03 00 02 00  |... .°â=x.......|
00000050  03 00 00 01 00 1b 00 53 6f 66 74 77 61 72 65 5c  |.......Software\|
00000060  43 6c 61 73 73 65 73 00 00 00 00 78 62 69 6e 30  |Classes....xbin0|
00000070  31 00 01 00 20 03 53 33 c0 b1 90 01 01 8a 98 90  |1... .S3À±......|
00000080  01 03 00 32 d9 88 98 90 01 03 00 40 3d 90 01 02  |...2Ù......@=...|
00000090  00 00 72 ea 90 00 01 00 34 03 6a 40 68 00 10 00  |..rê....4.j@h...|
000000a0  00 68 90 01 02 00 00 6a 00 ff d3 8b f0 56 68 90  |.h.....j.ÿÓ.ðVh.|
000000b0  01 02 00 00 68 90 01 02 40 00 e8 67 fa ff ff 8b  |....h...@.ègúÿÿ.|
000000c0  f8 6a 40 68 00 10 00 00 57 6a 00 ff 90 00 00 00  |øj@h....Wj.ÿ....|
000000d0  80 10 00 00 92 8d 9e 4e ff 76 d8 25 00 69 2a 9b  |.......NÿvØ%.i*.|
000000e0  00 10 00 80 5d 04 00 00 5f be 02 80 5c 1f 00 00  |....]..._¾..\...|
```

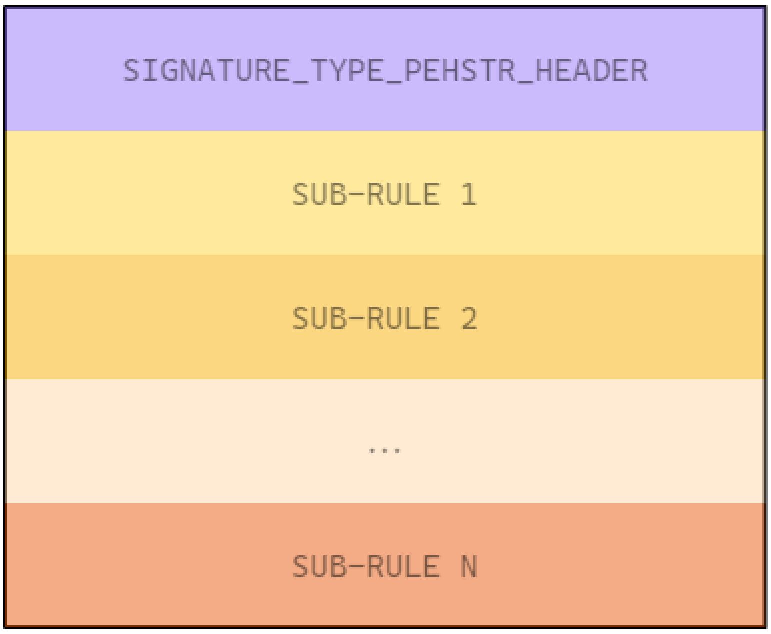
SIGNATURE_TYPE_PEHSTR vs SIGNATURE_TYPE_PEHSTR_EXT

The `SIGNATURE_TYPE_PEHSTR` and `SIGNATURE_TYPE_PEHSTR_EXT` are used to detect malicious Portable Executable (PE) files, where detection is based solely on byte and string matching.

Both of these signature types are composed of:

- A header
- One or more sub-rules

Each **sub-rule** has a specific **weight**, and the sum of all the matching sub-rules' weights must be greater than or equal to the rule's threshold value to trigger the detection.



To implement this mechanism, both SIGNATURE_TYPE_PEHSTR and SIGNATURE_TYPE_PEHSTR_EXT share a common header structure:

```
typedef struct _STRUCT_PEHSTR_HEADER {
  UINT16  ui16Unknown;
  UINT8   ui8ThresholdRequiredLow;
  UINT8   ui8ThresholdRequiredHigh;
  UINT8   ui8SubRulesNumberLow;
  UINT8   ui8SubRulesNumberHigh;
  BYTE    bEmpty;
  BYTE    pbRuleData[];
} STRUCT_PEHSTR_HEADER, * PSTRUCT_PEHSTR_HEADER;
```

where:

- ui16Unknown: the purpose of this field is unknown.
- ui8ThresholdRequiredLow: the low part of the threshold required to trigger a detection.
- ui8ThresholdRequiredHigh: the high part of the threshold required to trigger a detection.
- ui8SubRulesNumberLow: the low part of the number of sub-rules that compose this signature.
- ui8SubRulesNumberHigh: the high part of the number of sub-rules that compose this signature.
- pbRuleData[]: contains all the sub-rules used for detection.

For SIGNATURE_TYPE_PEHSTR_EXT, the sub-rules have the following structure:

```
typedef struct _STRUCT_RULE_PEHSTR_EXT {
  UINT8  ui8SubRuleWeightLow;
  UINT8  ui8SubRuleWeightHigh;
  UINT8  ui8SubRuleSize;
  UINT8  ui8CodeUnknown;
  BYTE   pbSubRuleBytesToMatch[];
} STRUCT_RULE_PEHSTR_EXT, *PSTRUCT_RULE_PEHSTR_EXT;
```

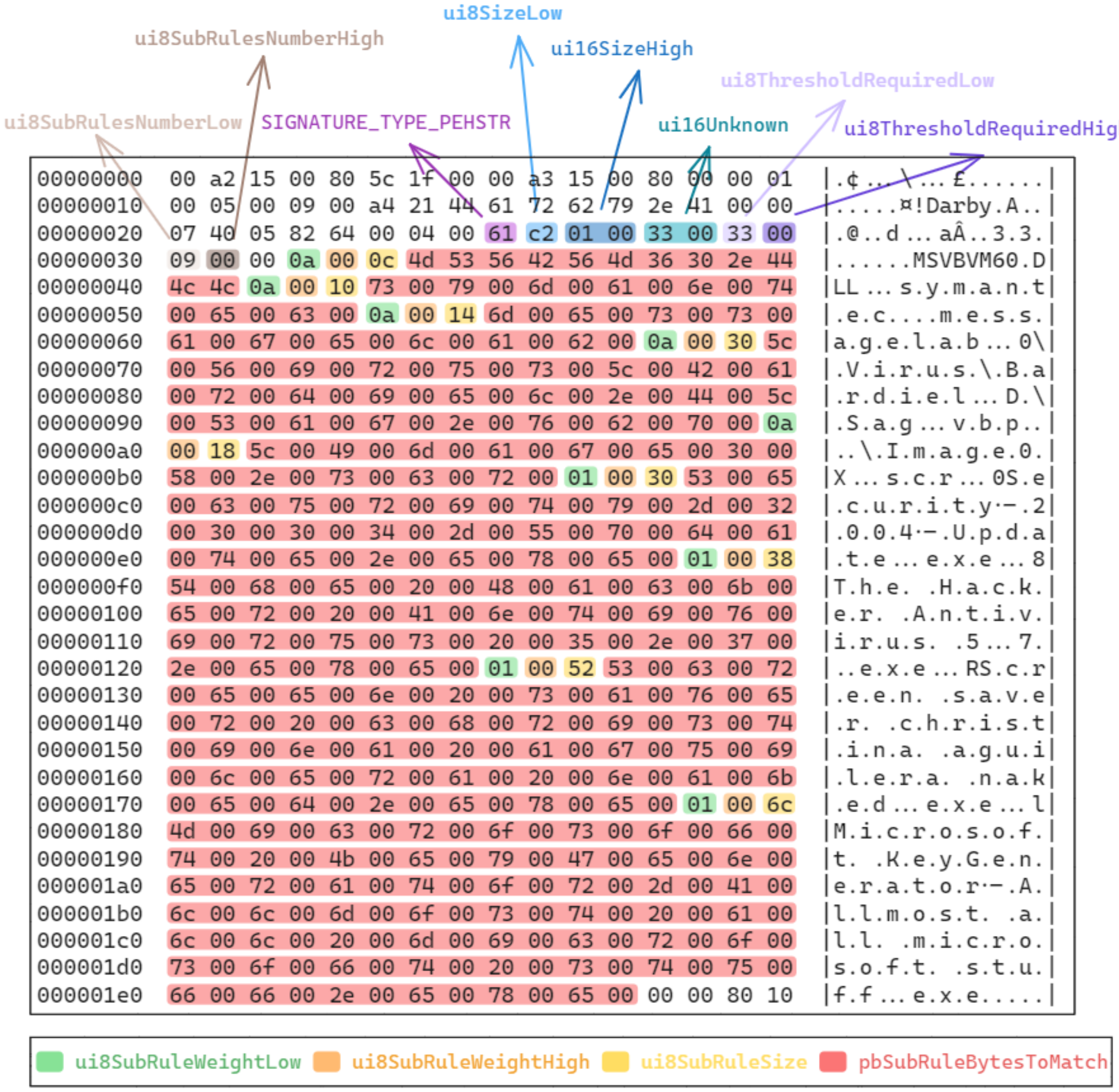
where:

- `ui8SubRuleWeightLow`: the low part of the weight of the sub-rule in the detection process.
- `ui8SubRuleWeightHigh`: the high part of the weight of the sub-rule in the detection process.
- `ui8SubRuleSize`: specifies the size of the byte string to match against a given PE.
- `pbSubRuleBytesToMatch[]`: the bytes that must be found to trigger a detection.

The `SIGNATURE_TYPE_PEHSTR` has the same structure except for the presence of `ui8CodeUnknown`.

Additionally, `SIGNATURE_TYPE_PEHSTR` can contain sub-rules with readable strings, while `SIGNATURE_TYPE_PEHSTR_EXT` can contain byte sequences.

The following image shows an example of a `SIGNATURE_TYPE_PEHSTR`:

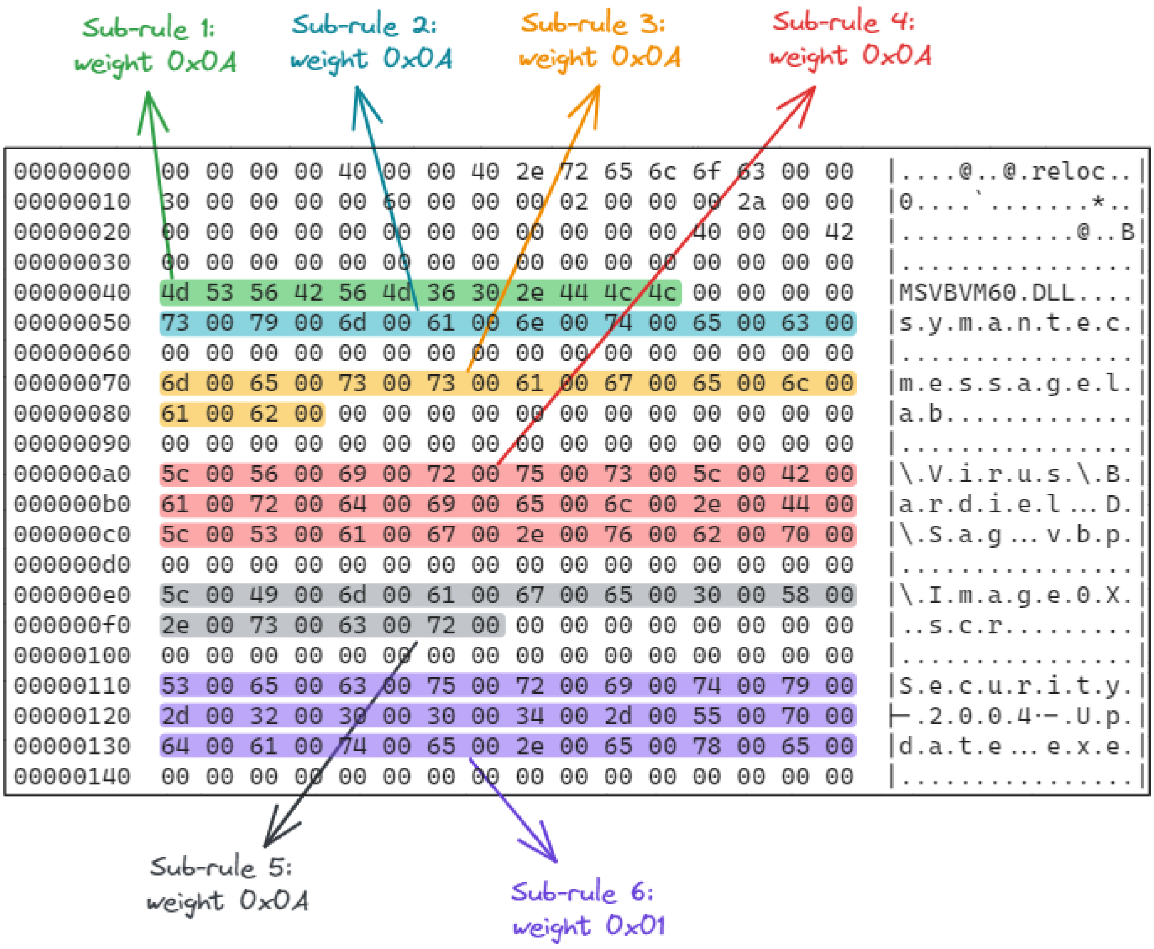


```
00000000  00 a2 15 00 80 5c 1f 00 00 a3 15 00 80 00 00 01  |.¢...\...£......|
00000010  00 05 00 09 00 a4 21 44 61 72 62 79 2e 41 00 00  |.....¤!Darby.A..|
00000020  07 40 05 82 64 00 04 00 61 c2 01 00 33 00 33 00  |.@..d...aÂ..3.3.|
00000030  09 00 00 0a 00 0c 4d 53 56 42 56 4d 36 30 2e 44  |......MSVBVM60.D|
00000040  4c 4c 0a 00 10 73 00 79 00 6d 00 61 00 6e 00 74  |LL...s.y.m.a.n.t|
00000050  00 65 00 63 00 0a 00 14 6d 00 65 00 73 00 73 00  |.e.c....m.e.s.s.|
00000060  61 00 67 00 65 00 6c 00 61 00 62 00 0a 00 30 5c  |a.g.e.l.a.b...0\|
00000070  00 56 00 69 00 72 00 75 00 73 00 5c 00 42 00 61  |.V.i.r.u.s.\.B.a|
00000080  00 72 00 64 00 69 00 65 00 6c 00 2e 00 44 00 5c  |.r.d.i.e.l...D.\|
00000090  00 53 00 61 00 67 00 2e 00 76 00 62 00 70 00 0a  |.S.a.g...v.b.p..|
000000a0  00 18 5c 00 49 00 6d 00 61 00 67 00 65 00 30 00  |..\.I.m.a.g.e.0.|
000000b0  58 00 2e 00 73 00 63 00 72 00 01 00 30 53 00 65  |X...s.c.r...0S.e|
000000c0  00 63 00 75 00 72 00 69 00 74 00 79 00 2d 00 32  |.c.u.r.i.t.y.-.2|
000000d0  00 30 00 30 00 34 00 2d 00 55 00 70 00 64 00 61  |.0.0.4.-.U.p.d.a|
000000e0  00 74 00 65 00 2e 00 65 00 78 00 65 00 01 00 38  |.t.e...e.x.e...8|
000000f0  54 00 68 00 65 00 20 00 48 00 61 00 63 00 6b 00  |T.h.e. .H.a.c.k.|
00000100  65 00 72 00 20 00 41 00 6e 00 74 00 69 00 76 00  |e.r. .A.n.t.i.v.|
00000110  69 00 72 00 75 00 73 00 20 00 35 00 2e 00 37 00  |i.r.u.s. .5...7.|
00000120  2e 00 65 00 78 00 65 00 01 00 52 53 00 63 00 72  |..e.x.e...RS.c.r|
00000130  00 65 00 65 00 6e 00 20 00 73 00 61 00 76 00 65  |.e.e.n. .s.a.v.e|
00000140  00 72 00 20 00 63 00 68 00 72 00 69 00 73 00 74  |.r. .c.h.r.i.s.t|
00000150  00 69 00 6e 00 61 00 20 00 61 00 67 00 75 00 69  |.i.n.a. .a.g.u.i|
00000160  00 6c 00 65 00 72 00 61 00 20 00 6e 00 61 00 6b  |.l.e.r.a. .n.a.k|
00000170  00 65 00 64 00 2e 00 65 00 78 00 65 00 01 00 6c  |.e.d...e.x.e...l|
00000180  4d 00 69 00 63 00 72 00 6f 00 73 00 6f 00 66 00  |M.i.c.r.o.s.o.f.|
00000190  74 00 20 00 4b 00 65 00 79 00 47 00 65 00 6e 00  |t. .K.e.y.G.e.n.|
000001a0  65 00 72 00 61 00 74 00 6f 00 72 00 2d 00 41 00  |e.r.a.t.o.r.-.A.|
000001b0  6c 00 6c 00 6d 00 6f 00 73 00 74 00 20 00 61 00  |l.l.m.o.s.t. .a.|
000001c0  6c 00 6c 00 20 00 6d 00 69 00 63 00 72 00 6f 00  |l.l. .m.i.c.r.o.|
000001d0  73 00 6f 00 66 00 74 00 20 00 73 00 74 00 75 00  |s.o.f.t. .s.t.u.|
000001e0  66 00 66 00 2e 00 65 00 78 00 65 00 00 00 80 10  |f.f...e.x.e.....|
```

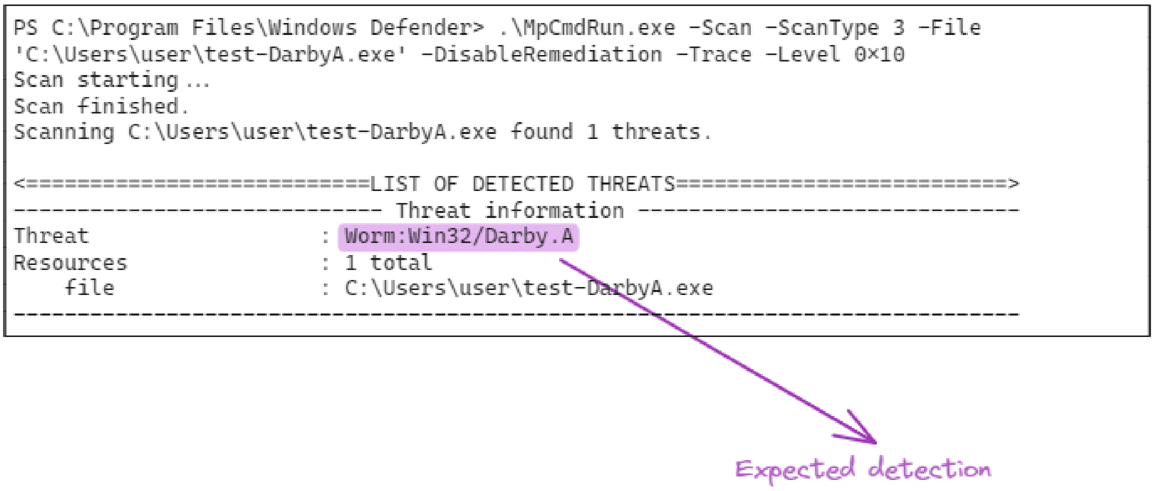
To trigger the detection of `Darby.A`, a threshold of at least **0x33** must be reached. The first five sub-rules have a weight (green field in the figure) of 0x0A, and the last four sub-rules have a weight of 0x01.

Any PE containing the bytes from the first five sub-rules (all with a weight of 0x0A) and at least one of the last four sub-rules (with a weight of 0x01) will meet the threshold of 0x33 and be detected.

Using the tool `MpCmdRun.exe`, the detection can be verified:



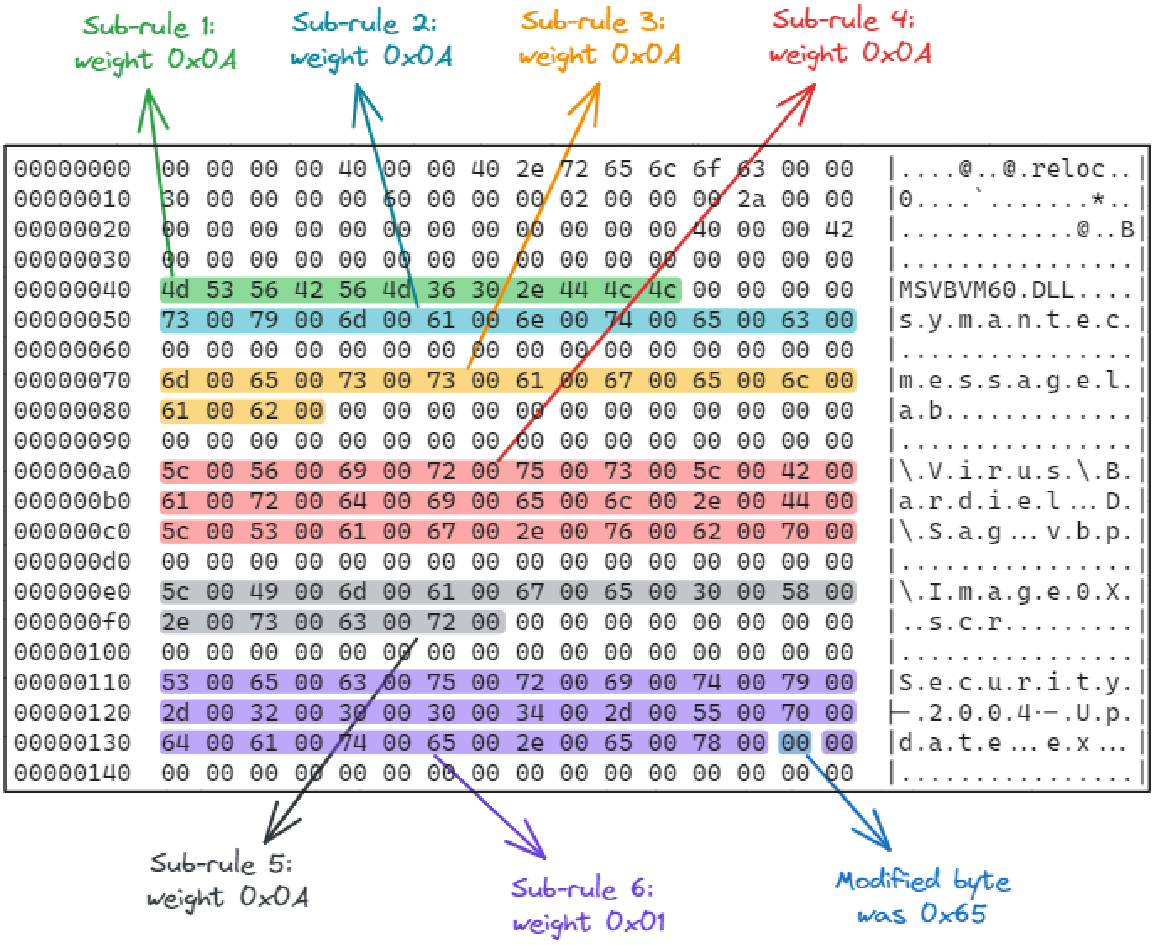
```
00000000  00 00 00 00 40 00 00 40 2e 72 65 6c 6f 63 00 00  |....@..@.reloc..|
00000010  30 00 00 00 00 60 00 00 00 02 00 00 00 2a 00 00  |0....`.......*..|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42  |............@..B|
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000040  4d 53 56 42 56 4d 36 30 2e 44 4c 4c 00 00 00 00  |MSVBVM60.DLL....|
00000050  73 00 79 00 6d 00 61 00 6e 00 74 00 65 00 63 00  |s.y.m.a.n.t.e.c.|
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000070  6d 00 65 00 73 00 73 00 61 00 67 00 65 00 6c 00  |m.e.s.s.a.g.e.l.|
00000080  61 00 62 00 00 00 00 00 00 00 00 00 00 00 00 00  |a.b.............|
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000a0  5c 00 56 00 69 00 72 00 75 00 73 00 5c 00 42 00  |\.V.i.r.u.s.\.B.|
000000b0  61 00 72 00 64 00 69 00 65 00 6c 00 2e 00 44 00  |a.r.d.i.e.l...D.|
000000c0  5c 00 53 00 61 00 67 00 2e 00 76 00 62 00 70 00  |\.S.a.g...v.b.p.|
000000d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000e0  5c 00 49 00 6d 00 61 00 67 00 65 00 30 00 58 00  |\.I.m.a.g.e.0.X.|
000000f0  2e 00 73 00 63 00 72 00 00 00 00 00 00 00 00 00  |..s.c.r.........|
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000110  53 00 65 00 63 00 75 00 72 00 69 00 74 00 79 00  |S.e.c.u.r.i.t.y.|
00000120  2d 00 32 00 30 00 30 00 34 00 2d 00 55 00 70 00  |-.2.0.0.4.-.U.p.|
00000130  64 00 61 00 74 00 65 00 2e 00 65 00 78 00 65 00  |d.a.t.e...e.x.e.|
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```



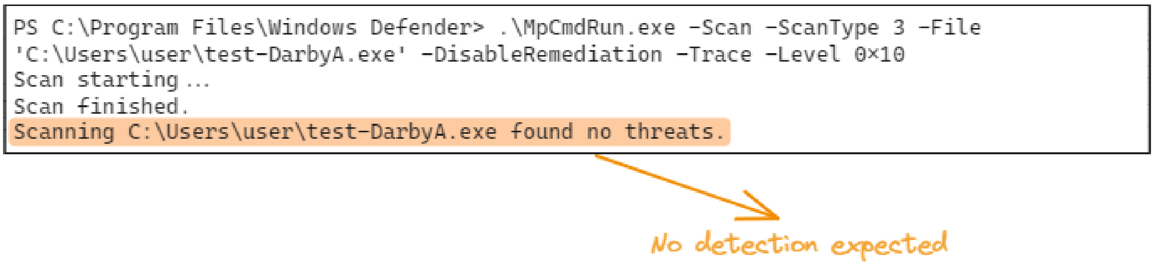
```
PS C:\Program Files\Windows Defender> .\MpCmdRun.exe -Scan -ScanType 3 -File
'C:\Users\user\test-DarbyA.exe' -DisableRemediation -Trace -Level 0x10
Scan starting...
Scan finished.
Scanning C:\Users\user\test-DarbyA.exe found 1 threats.

<============================LIST OF DETECTED THREATS===========================>
----------------------------- Threat information ------------------------------
Threat                  : Worm:Win32/Darby.A
Resources               : 1 total
    file                : C:\Users\user\test-DarbyA.exe
--------------------------------------------------------------------------------
```

If even **one single byte** of a sub-rule is missing from the PE, detection will not occur. In the following picture, the modified byte is highlighted in blue:



```
00000000  00 00 00 00 40 00 00 40 2e 72 65 6c 6f 63 00 00  |....@..@.reloc..|
00000010  30 00 00 00 00 60 00 00 00 02 00 00 00 2a 00 00  |0....`.......*..|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42  |............@..B|
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000040  4d 53 56 42 56 4d 36 30 2e 44 4c 4c 00 00 00 00  |MSVBVM60.DLL....|
00000050  73 00 79 00 6d 00 61 00 6e 00 74 00 65 00 63 00  |s.y.m.a.n.t.e.c.|
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000070  6d 00 65 00 73 00 73 00 61 00 67 00 65 00 6c 00  |m.e.s.s.a.g.e.l.|
00000080  61 00 62 00 00 00 00 00 00 00 00 00 00 00 00 00  |a.b.............|
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000a0  5c 00 56 00 69 00 72 00 75 00 73 00 5c 00 42 00  |\.V.i.r.u.s.\.B.|
000000b0  61 00 72 00 64 00 69 00 65 00 6c 00 2e 00 44 00  |a.r.d.i.e.l...D.|
000000c0  5c 00 53 00 61 00 67 00 2e 00 76 00 62 00 70 00  |\.S.a.g...v.b.p.|
000000d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000e0  5c 00 49 00 6d 00 61 00 67 00 65 00 30 00 58 00  |\.I.m.a.g.e.0.X.|
000000f0  2e 00 73 00 63 00 72 00 00 00 00 00 00 00 00 00  |..s.c.r.........|
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000110  53 00 65 00 63 00 75 00 72 00 69 00 74 00 79 00  |S.e.c.u.r.i.t.y.|
00000120  2d 00 32 00 30 00 30 00 34 00 2d 00 55 00 70 00  |-.2.0.0.4.-.U.p.|
00000130  64 00 61 00 74 00 65 00 2e 00 65 00 78 00 00 00  |d.a.t.e...e.x...|
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```



```
PS C:\Program Files\Windows Defender> .\MpCmdRun.exe -Scan -ScanType 3 -File
'C:\Users\user\test-DarbyA.exe' -DisableRemediation -Trace -Level 0x10
Scan starting...
Scan finished.
Scanning C:\Users\user\test-DarbyA.exe found no threats.
```

However, it's important to note that such rules, based solely on byte and string matching, are relatively easy to bypass. For that reason, a lot of wildcards have been introduced to make the rules stronger and more flexible.

Pattern to implement wildcards

`0x90`, followed by a second byte that identifies its "type", and then more bytes that define it. Patterns from `90 01` to `90 20` are used to implement wildcards within the `SIGNATURE_TYPE_PEHSTR_EXT` parsing algorithm.

The following patterns will be explained in the next sections:

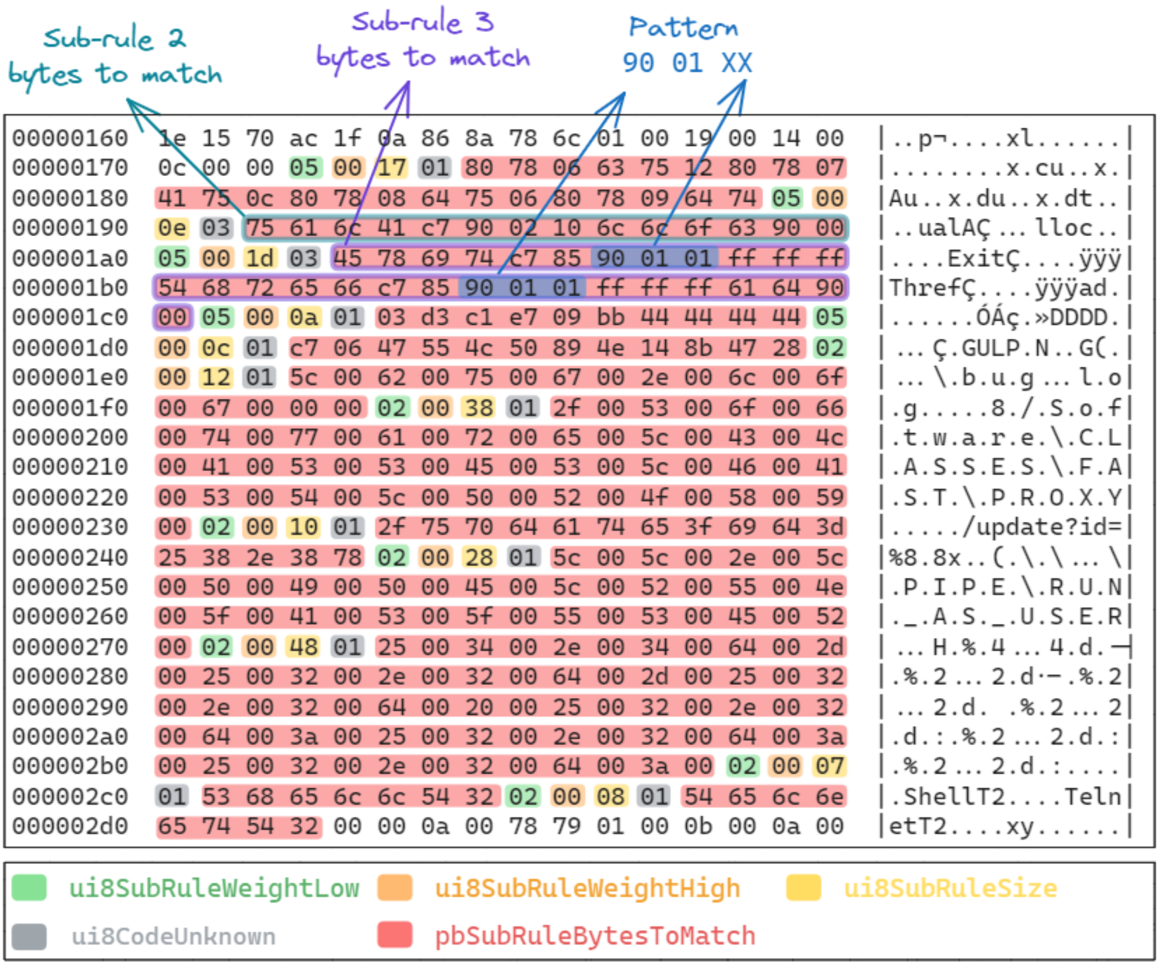
- `90 01 XX`
- `90 02 XX`
- `90 03 XX YY`

- 90 04 XX YY
- 90 05 XX YY

Pattern 90 01 XX

The pattern 90 01 XX is used to match a sequence of bytes of a specific length, defined by the quantity XX.

In the next figure, the sub-rules related to the Plugx.A signature are shown:

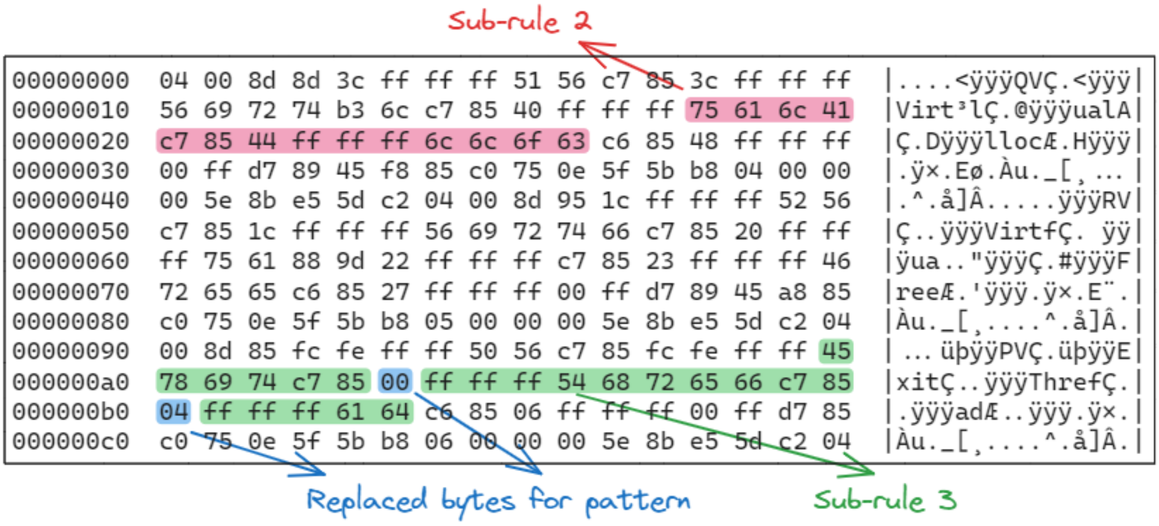


```
00000160  1e 15 70 ac 1f 0a 86 8a 78 6c 01 00 19 00 14 00  |..p¬....xl......|
00000170  0c 00 00 05 00 17 01 80 78 06 63 75 12 80 78 07  |........x.cu..x.|
00000180  41 75 0c 80 78 08 64 75 06 80 78 09 64 74 05 00  |Au..x.du..x.dt..|
00000190  0e 03 75 61 6c 41 c7 90 02 10 6c 6c 6f 63 90 00  |..ualAÇ...lloc..|
000001a0  05 00 1d 03 45 78 69 74 c7 85 90 01 01 ff ff ff  |....ExitÇ....ÿÿÿ|
000001b0  54 68 72 65 66 c7 85 90 01 01 ff ff ff 61 64 90  |ThrefÇ....ÿÿÿad.|
000001c0  00 05 00 0a 01 03 d3 c1 e7 09 bb 44 44 44 44 05  |......ÓÁç.»DDDD.|
000001d0  00 0c 01 c7 06 47 55 4c 50 89 4e 14 8b 47 28 02  |...Ç.GULP.N..G(.|
000001e0  00 12 01 5c 00 62 00 75 00 67 00 2e 00 6c 00 6f  |...\.b.u.g...l.o|
000001f0  00 67 00 00 00 02 00 38 01 2f 00 53 00 6f 00 66  |.g.....8./.S.o.f|
00000200  00 74 00 77 00 61 00 72 00 65 00 5c 00 43 00 4c  |.t.w.a.r.e.\.C.L|
00000210  00 41 00 53 00 53 00 45 00 53 00 5c 00 46 00 41  |.A.S.S.E.S.\.F.A|
00000220  00 53 00 54 00 5c 00 50 00 52 00 4f 00 58 00 59  |.S.T.\.P.R.O.X.Y|
00000230  00 02 00 10 01 2f 75 70 64 61 74 65 3f 69 64 3d  |...../update?id=|
00000240  25 38 2e 38 78 02 00 28 01 5c 00 5c 00 2e 00 5c  |%8.8x..(.\.\...\|
00000250  00 50 00 49 00 50 00 45 00 5c 00 52 00 55 00 4e  |.P.I.P.E.\.R.U.N|
00000260  00 5f 00 41 00 53 00 5f 00 55 00 53 00 45 00 52  |._.A.S._.U.S.E.R|
00000270  00 02 00 48 01 25 00 34 00 2e 00 34 00 64 00 2d  |...H.%.4...4.d.-|
00000280  00 25 00 32 00 2e 00 32 00 64 00 2d 00 25 00 32  |.%.2...2.d·-.%.2|
00000290  00 2e 00 32 00 64 00 20 00 25 00 32 00 2e 00 32  |...2.d. .%.2...2|
000002a0  00 64 00 3a 00 25 00 32 00 2e 00 32 00 64 00 3a  |.d.:.%.2...2.d.:|
000002b0  00 25 00 32 00 2e 00 32 00 64 00 3a 00 02 00 07  |.%.2...2.d.:....|
000002c0  01 53 68 65 6c 6c 54 32 02 00 08 01 54 65 6c 6e  |.ShellT2....Teln|
000002d0  65 74 54 32 00 00 0a 00 78 79 01 00 0b 00 0a 00  |etT2....xy......|
```

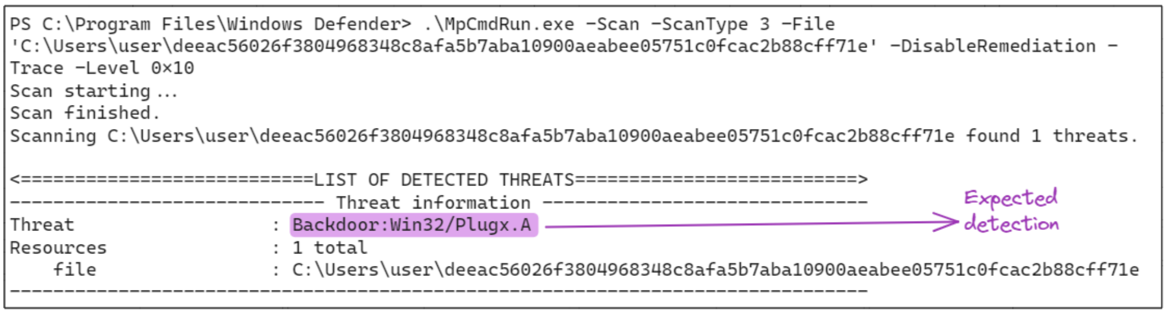
🟩 ui8SubRuleWeightLow 🟧 ui8SubRuleWeightHigh 🟨 ui8SubRuleSize

⬜ ui8CodeUnknown 🟥 pbSubRuleBytesToMatch

There are two sequences of the pattern `90 01 01`, highlighted in blue within the same sub-rule. In both positions, exactly one byte of any value is expected.
An example of detection for the rule depicted in **Figure 14**.



```
00000000  04 00 8d 8d 3c ff ff ff 51 56 c7 85 3c ff ff ff  |....<ÿÿÿQVÇ.<ÿÿÿ|
00000010  56 69 72 74 b3 6c c7 85 40 ff ff ff 75 61 6c 41  |Virt³lÇ.@ÿÿÿualA|
00000020  c7 85 44 ff ff ff 6c 6c 6f 63 c6 85 48 ff ff ff  |Ç.DÿÿÿllocÆ.Hÿÿÿ|
00000030  00 ff d7 89 45 f8 85 c0 75 0e 5f 5b b8 04 00 00  |.ÿ×.Eø.Àu._[¸...|
00000040  00 5e 8b e5 5d c2 04 00 8d 95 1c ff ff ff 52 56  |.^.å]Â.....ÿÿÿRV|
00000050  c7 85 1c ff ff ff 56 69 72 74 66 c7 85 20 ff ff  |Ç..ÿÿÿVirtfÇ. ÿÿ|
00000060  ff 75 61 88 9d 22 ff ff ff c7 85 23 ff ff ff 46  |ÿua.."ÿÿÿÇ.#ÿÿÿF|
00000070  72 65 65 c6 85 27 ff ff ff 00 ff d7 89 45 a8 85  |reeÆ.'ÿÿÿ.ÿ×.E¨.|
00000080  c0 75 0e 5f 5b b8 05 00 00 00 5e 8b e5 5d c2 04  |Àu._[¸....^.å]Â.|
00000090  00 8d 85 fc fe ff ff 50 56 c7 85 fc fe ff ff 45  |...üþÿÿPVÇ.üþÿÿE|
000000a0  78 69 74 c7 85 00 ff ff ff 54 68 72 65 66 c7 85  |xitÇ..ÿÿÿThrefÇ.|
000000b0  04 ff ff ff 61 64 c6 85 06 ff ff ff 00 ff d7 85  |.ÿÿÿadÆ..ÿÿÿ.ÿ×.|
000000c0  c0 75 0e 5f 5b b8 06 00 00 00 5e 8b e5 5d c2 04  |Àu._[¸....^.å]Â.|
```



```
PS C:\Program Files\Windows Defender> .\MpCmdRun.exe -Scan -ScanType 3 -File
'C:\Users\user\deeac56026f3804968348c8afa5b7aba10900aeabee05751c0fcac2b88cff71e' -DisableRemediation -
Trace -Level 0x10
Scan starting...
Scan finished.
Scanning C:\Users\user\deeac56026f3804968348c8afa5b7aba10900aeabee05751c0fcac2b88cff71e found 1 threats.

<===========================LIST OF DETECTED THREATS==========================>
----------------------------- Threat information ------------------------------
Threat                  : Backdoor:Win32/Plugx.A
Resources               : 1 total
    file                : C:\Users\user\deeac56026f3804968348c8afa5b7aba10900aeabee05751c0fcac2b88cff71e
-------------------------------------------------------------------------------
```

The Yara representation for the sub-rule can be written as follows:

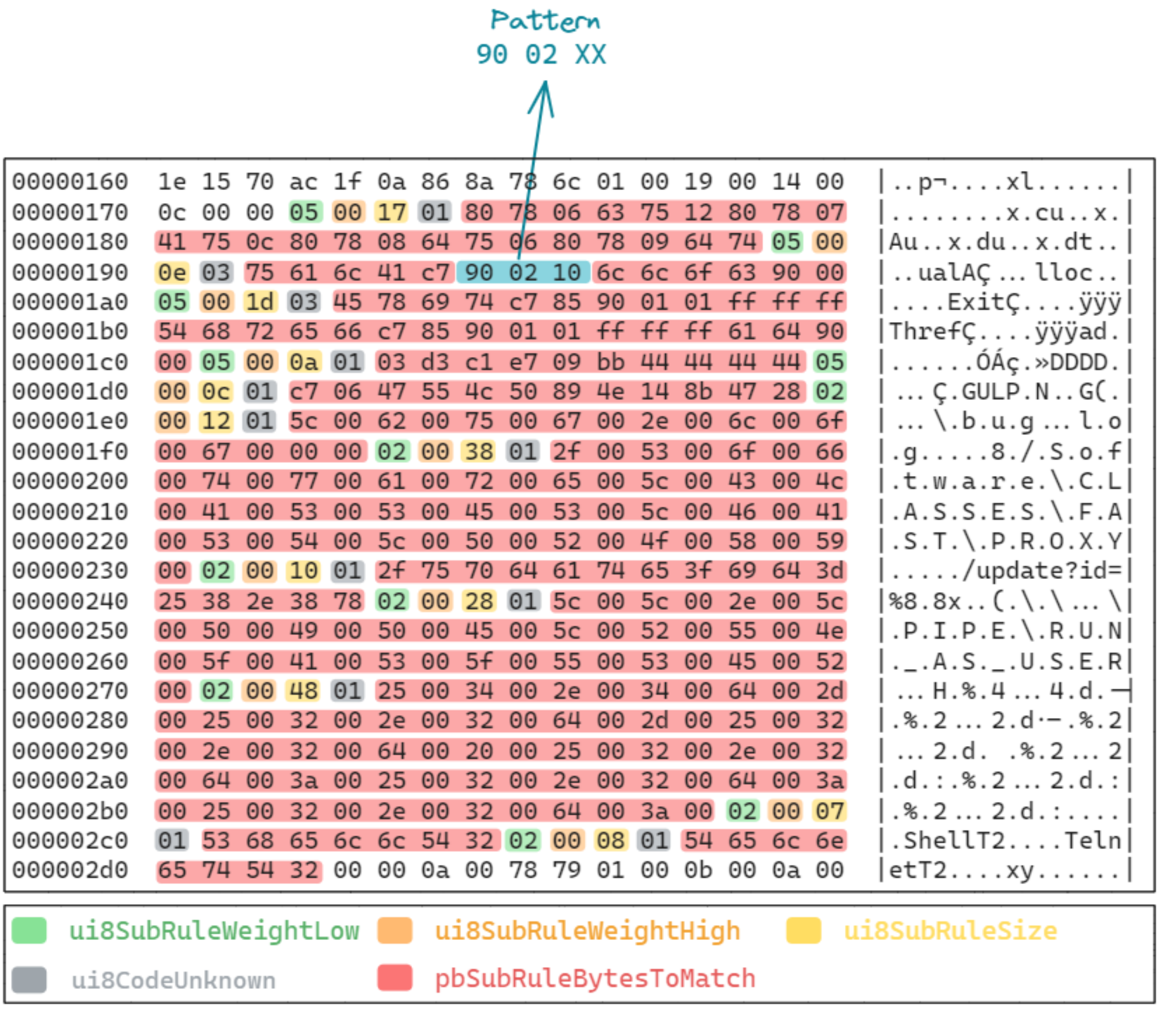
```
rule Pattern_90_01_example
{
    strings:
       $sub_rule_3_hex = { 45 78 69 74 C7 85 ?? FF FF FF 54 68 72 65 66 C7 85 ??
04 FF FF FF 61 64 }

    condition:
       $sub_rule_3_hex
}
```

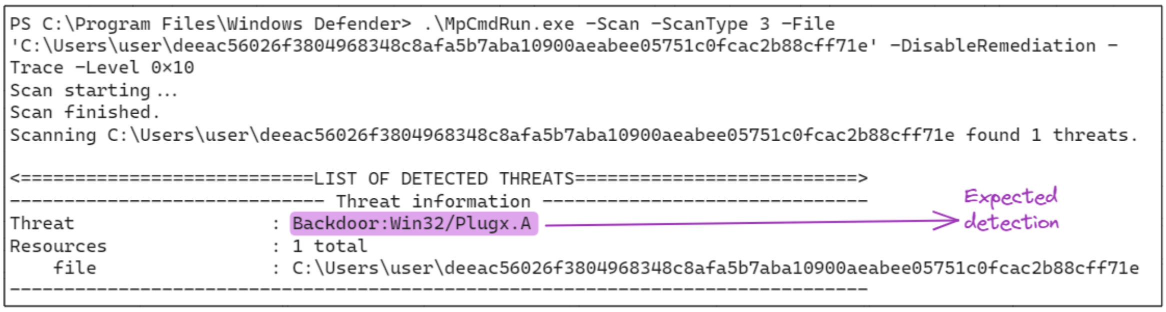
Pattern 90 02 XX

The pattern `90 02 XX` is used as a placeholder to match **up to** `XX` bytes in a specific position.

As with the pattern `90 01 XX`, an example of the pattern `90 02 XX` from a sub-rule found within the `Plugx.A` threat is shown in **Figure 15**: The pattern identified by `90 02 10` matches up to **16** bytes in that position.



```
00000160  1e 15 70 ac 1f 0a 86 8a 78 6c 01 00 19 00 14 00  |..p¬....xl......|
00000170  0c 00 00 05 00 17 01 80 78 06 63 75 12 80 78 07  |........x.cu..x.|
00000180  41 75 0c 80 78 08 64 75 06 80 78 09 64 74 05 00  |Au..x.du..x.dt..|
00000190  0e 03 75 61 6c 41 c7 90 02 10 6c 6c 6f 63 90 00  |..ualAÇ...lloc..|
000001a0  05 00 1d 03 45 78 69 74 c7 85 90 01 01 ff ff ff  |....ExitÇ....ÿÿÿ|
000001b0  54 68 72 65 66 c7 85 90 01 01 ff ff ff 61 64 90  |ThrefÇ....ÿÿÿad.|
000001c0  00 05 00 0a 01 03 d3 c1 e7 09 bb 44 44 44 44 05  |......ÓÁç.»DDDD.|
000001d0  00 0c 01 c7 06 47 55 4c 50 89 4e 14 8b 47 28 02  |...Ç.GULP.N..G(.|
000001e0  00 12 01 5c 00 62 00 75 00 67 00 2e 00 6c 00 6f  |...\.b.u.g...l.o|
000001f0  00 67 00 00 00 02 00 38 01 2f 00 53 00 6f 00 66  |.g.....8./.S.o.f|
00000200  00 74 00 77 00 61 00 72 00 65 00 5c 00 43 00 4c  |.t.w.a.r.e.\.C.L|
00000210  00 41 00 53 00 53 00 45 00 53 00 5c 00 46 00 41  |.A.S.S.E.S.\.F.A|
00000220  00 53 00 54 00 5c 00 50 00 52 00 4f 00 58 00 59  |.S.T.\.P.R.O.X.Y|
00000230  00 02 00 10 01 2f 75 70 64 61 74 65 3f 69 64 3d  |...../update?id=|
00000240  25 38 2e 38 78 02 00 28 01 5c 00 5c 00 2e 00 5c  |%8.8x..(.\.\...\|
00000250  00 50 00 49 00 50 00 45 00 5c 00 52 00 55 00 4e  |.P.I.P.E.\.R.U.N|
00000260  00 5f 00 41 00 53 00 5f 00 55 00 53 00 45 00 52  |._.A.S._.U.S.E.R|
00000270  00 02 00 48 01 25 00 34 00 2e 00 34 00 64 00 2d  |...H.%.4...4.d.─|
00000280  00 25 00 32 00 2e 00 32 00 64 00 2d 00 25 00 32  |.%.2...2.d·─.%.2|
00000290  00 2e 00 32 00 64 00 20 00 25 00 32 00 2e 00 32  |...2.d. .%.2...2|
000002a0  00 64 00 3a 00 25 00 32 00 2e 00 32 00 64 00 3a  |.d.:.%.2...2.d.:|
000002b0  00 25 00 32 00 2e 00 32 00 64 00 3a 00 02 00 07  |.%.2...2.d.:....|
000002c0  01 53 68 65 6c 6c 54 32 02 00 08 01 54 65 6c 6e  |.ShellT2....Teln|
000002d0  65 74 54 32 00 00 0a 00 78 79 01 00 0b 00 0a 00  |etT2....xy......|
```



```
PS C:\Program Files\Windows Defender> .\MpCmdRun.exe -Scan -ScanType 3 -File
'C:\Users\user\deeac56026f3804968348c8afa5b7aba10900aeabee05751c0fcac2b88cff71e' -DisableRemediation -
Trace -Level 0x10
Scan starting...
Scan finished.
Scanning C:\Users\user\deeac56026f3804968348c8afa5b7aba10900aeabee05751c0fcac2b88cff71e found 1 threats.

<===========================LIST OF DETECTED THREATS==========================>
----------------------------- Threat information ------------------------------
Threat                      : Backdoor:Win32/Plugx.A
Resources                   : 1 total
    file                    : C:\Users\user\deeac56026f3804968348c8afa5b7aba10900aeabee05751c0fcac2b88cff71e
-------------------------------------------------------------------------------
```

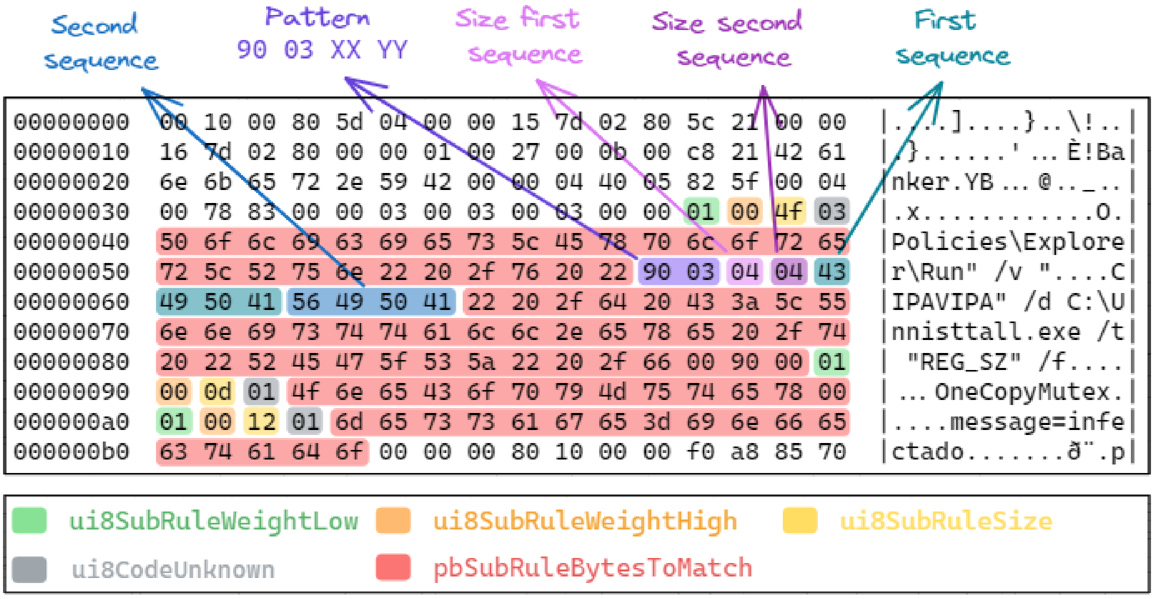
The Yara counterpart for the above sub-rule can be written as follows:

```
rule Pattern_90_02_example
{
     strings:
        $sub_rule_2_hex = { 75 61 6C 41 C7 [0-16] 6C 6C 6F 63 }

     condition:
        $sub_rule_2_hex
}
```

Pattern 90 03 XX YY

The pattern `90 03 XX YY` is followed by two consecutive sequences of bytes whose lengths are defined by `XX` and `YY`. The expected bytes to be found must match one of the two sequences. An example is depicted in **Figure 17**:



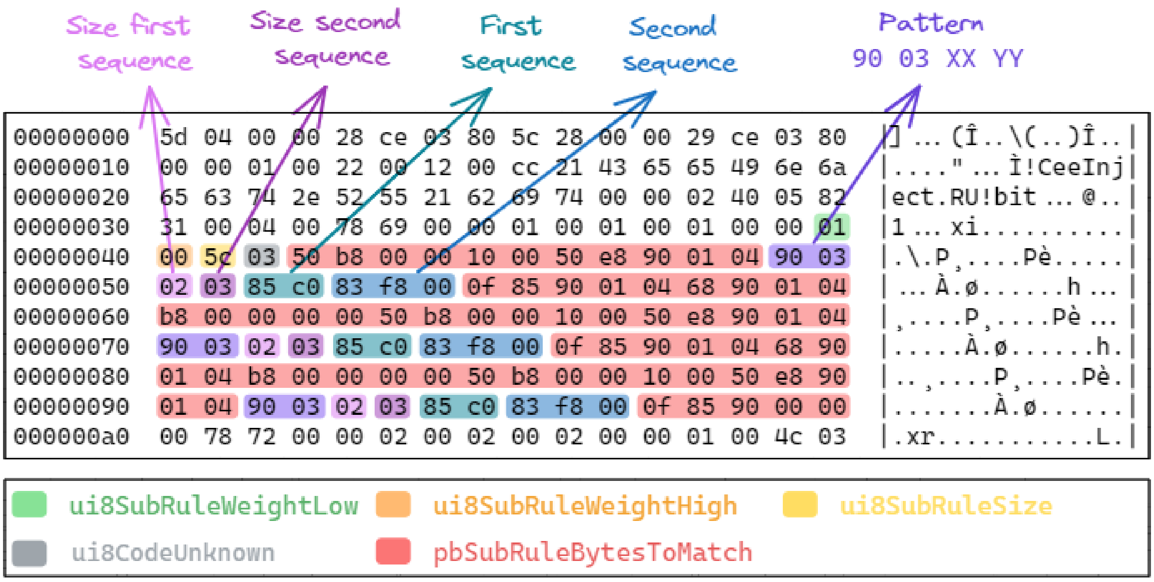
```
00000000  00 10 00 80 5d 04 00 00 15 7d 02 80 5c 21 00 00  |....]....}..\!..|
00000010  16 7d 02 80 00 00 01 00 27 00 0b 00 c8 21 42 61  |.}......'...È!Ba|
00000020  6e 6b 65 72 2e 59 42 00 00 04 40 05 82 5f 00 04  |nker.YB...@.._..|
00000030  00 78 83 00 00 03 00 03 00 03 00 00 01 00 4f 03  |.x............O.|
00000040  50 6f 6c 69 63 69 65 73 5c 45 78 70 6c 6f 72 65  |Policies\Explore|
00000050  72 5c 52 75 6e 22 20 2f 76 20 22 90 03 04 04 43  |r\Run" /v "....C|
00000060  49 50 41 56 49 50 41 22 20 2f 64 20 43 3a 5c 55  |IPAVIPA" /d C:\U|
00000070  6e 6e 69 73 74 74 61 6c 6c 2e 65 78 65 20 2f 74  |nnisttall.exe /t|
00000080  20 22 52 45 47 5f 53 5a 22 20 2f 66 00 90 00 01  | "REG_SZ" /f....|
00000090  00 0d 01 4f 6e 65 43 6f 70 79 4d 75 74 65 78 00  |...OneCopyMutex.|
000000a0  01 00 12 01 6d 65 73 73 61 67 65 3d 69 6e 66 65  |....message=infe|
000000b0  63 74 61 64 6f 00 00 00 80 10 00 00 f0 a8 85 70  |ctado.......ð¨.p|
```

🟩 ui8SubRuleWeightLow 🟧 ui8SubRuleWeightHigh 🟨 ui8SubRuleSize

⬜ ui8CodeUnknown 🟥 pbSubRuleBytesToMatch

In **Figure 16**, the example is related to the `Banker.YB` threat, where the pattern allows the choice of one of two strings, **"CIPA"** or **"VIPA"**.

The pattern `90 03 XX YY` can also deal with general byte sequences, as shown in **Figure 17**:



```
00000000  5d 04 00 00 28 ce 03 80 5c 28 00 00 29 ce 03 80  |]...(Î..\(..)Î..|
00000010  00 00 01 00 22 00 12 00 cc 21 43 65 65 49 6e 6a  |...."...Ì!CeeInj|
00000020  65 63 74 2e 52 55 21 62 69 74 00 00 02 40 05 82  |ect.RU!bit...@..|
00000030  31 00 04 00 78 69 00 00 01 00 01 00 01 00 00 01  |1...xi..........|
00000040  00 5c 03 50 b8 00 00 10 00 50 e8 90 01 04 90 03  |.\.P¸....Pè.....|
00000050  02 03 85 c0 83 f8 00 0f 85 90 01 04 68 90 01 04  |...À.ø......h...|
00000060  b8 00 00 00 00 50 b8 00 00 10 00 50 e8 90 01 04  |¸....P¸....Pè...|
00000070  90 03 02 03 85 c0 83 f8 00 0f 85 90 01 04 68 90  |.....À.ø......h.|
00000080  01 04 b8 00 00 00 00 50 b8 00 00 10 00 50 e8 90  |..¸....P¸....Pè.|
00000090  01 04 90 03 02 03 85 c0 83 f8 00 0f 85 90 00 00  |.......À.ø......|
000000a0  00 78 72 00 00 02 00 02 00 02 00 00 01 00 4c 03  |.xr...........L.|
```

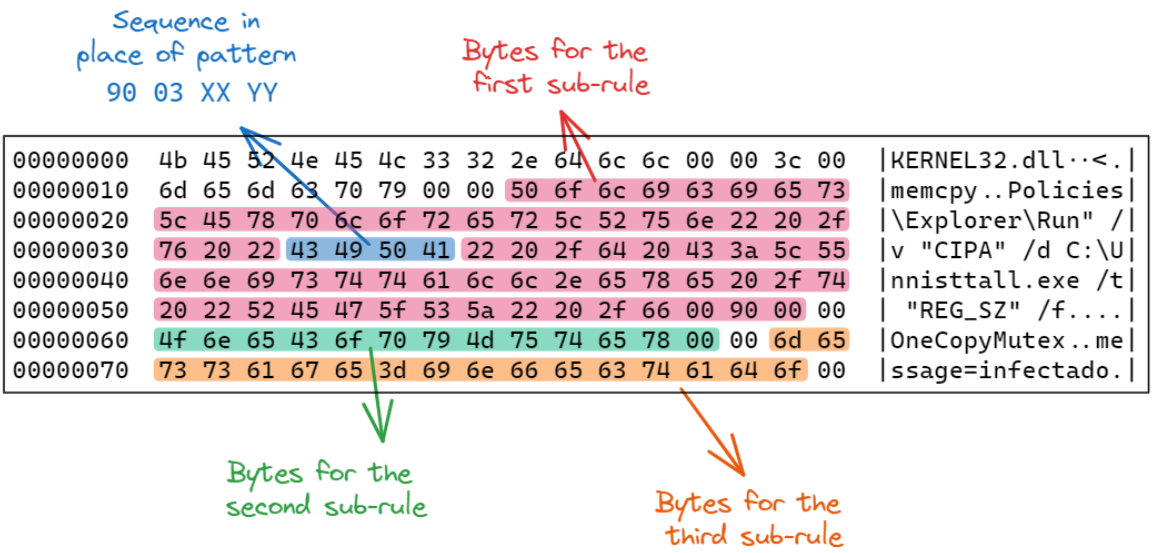
🟩 ui8SubRuleWeightLow 🟧 ui8SubRuleWeightHigh 🟨 ui8SubRuleSize

⬜ ui8CodeUnknown 🟥 pbSubRuleBytesToMatch

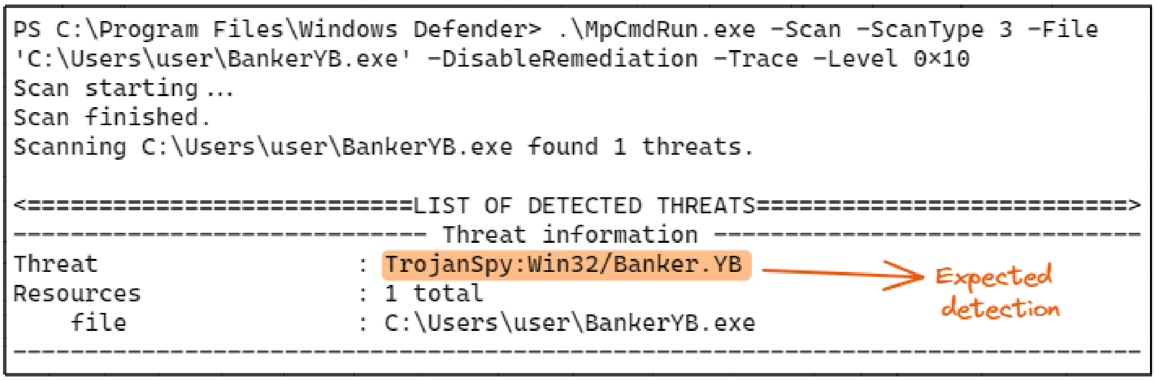
The pattern can be described using the following Yara rule:

```
rule Pattern_90_03_example
{
    strings:
       $sub_rule_1_hex = { 50 6f 6c 69 63 69 65 73 5c 45 78 70 6c 6f 72 65 72 5c
52 75 6e 22 20 2f 76 20 22 (43 49 50 41|56 49 50 41) 22 20 2f 64 20 43 3a 5c 55
6e 6e 69 73 74 74 61 6c 6c 2e 65 78 65 20 2f 74 20 22 52 45 47 5f 53 5a 22 20 2f
66 00 90 00 }
    condition:
       $sub_rule_1_hex
}
```

The following PE part will trigger the `Banker.YB` detection:


Sequence in place of pattern 90 03 XX YY
Bytes for the first sub-rule
Bytes for the second sub-rule
Bytes for the third sub-rule

```
00000000  4b 45 52 4e 45 4c 33 32 2e 64 6c 6c 00 00 3c 00  |KERNEL32.dll··<.|
00000010  6d 65 6d 63 70 79 00 00 50 6f 6c 69 63 69 65 73  |memcpy..Policies|
00000020  5c 45 78 70 6c 6f 72 65 72 5c 52 75 6e 22 20 2f  |\Explorer\Run" /|
00000030  76 20 22 43 49 50 41 22 20 2f 64 20 43 3a 5c 55  |v "CIPA" /d C:\U|
00000040  6e 6e 69 73 74 74 61 6c 6c 2e 65 78 65 20 2f 74  |nnisttall.exe /t|
00000050  20 22 52 45 47 5f 53 5a 22 20 2f 66 00 90 00 00  | "REG_SZ" /f....|
00000060  4f 6e 65 43 6f 70 79 4d 75 74 65 78 00 00 6d 65  |OneCopyMutex..me|
00000070  73 73 61 67 65 3d 69 6e 66 65 63 74 61 64 6f 00  |ssage=infectado.|
```

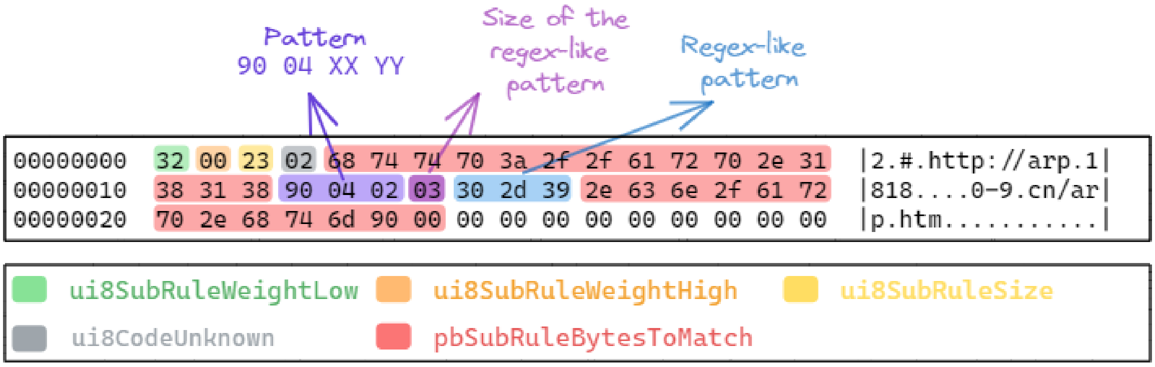


```
PS C:\Program Files\Windows Defender> .\MpCmdRun.exe -Scan -ScanType 3 -File
'C:\Users\user\BankerYB.exe' -DisableRemediation -Trace -Level 0x10
Scan starting...
Scan finished.
Scanning C:\Users\user\BankerYB.exe found 1 threats.

<===========================LIST OF DETECTED THREATS==========================>
----------------------------- Threat information ------------------------------
Threat                  : TrojanSpy:Win32/Banker.YB
Resources               : 1 total
    file                : C:\Users\user\BankerYB.exe
-------------------------------------------------------------------------------
```

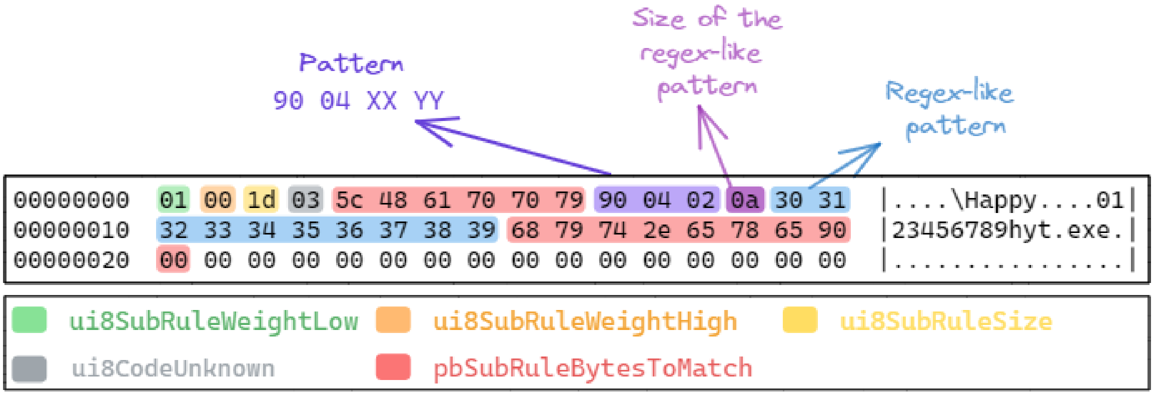
Pattern 90 04 XX YY

The pattern `90 04 XX YY` is a placeholder for a regex-like expression, where `XX` represents the **exact number of bytes** that must be found, and `YY` represents the length of the regex-like pattern. An example of this pattern can be found in **Figure 19**:



```
00000000  32 00 23 02 68 74 74 70 3a 2f 2f 61 72 70 2e 31  |2.#.http://arp.1|
00000010  38 31 38 90 04 02 03 30 2d 39 2e 63 6e 2f 61 72  |818....0-9.cn/ar|
00000020  70 2e 68 74 6d 90 00 00 00 00 00 00 00 00 00 00  |p.htm...........|
```

More complex patterns can contain the exact characters to match:



```
00000000  01 00 1d 03 5c 48 61 70 70 79 90 04 02 0a 30 31  |....\Happy....01|
00000010  32 33 34 35 36 37 38 39 68 79 74 2e 65 78 65 90  |23456789hyt.exe.|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

In **Figure 19** and **Figure 20**, the pattern itself is highlighted in violet, the size of the regex-like pattern in grape, and the regex-like bytes in blue.

The patterns can be represented as Yara rules:

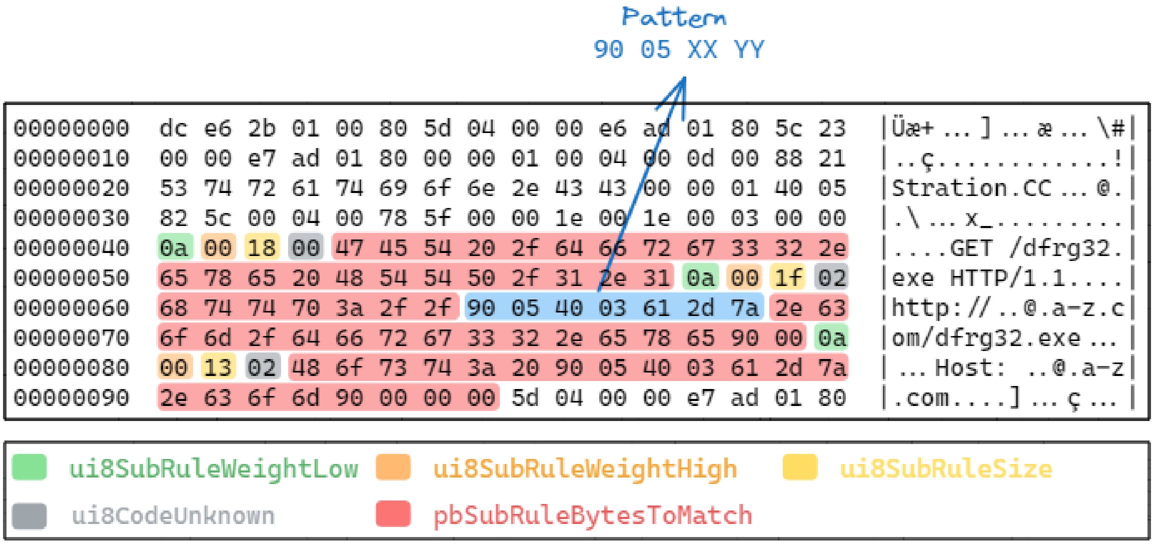
```
rule Pattern_90_04_example
{
    strings:
       $example_90_04_first_rule = { 68 74 74 70 3a 2f 2f 61 72 70 2e 31 38 31 38
[30-39] [30-39] 2e 63 6e 2f 61 72 70 2e 68 74 6d 90 00 }
       $example_90_04_second_rule = { 5c 48 61 70 70 79 [30-39] [30-39] 68 79 74
2e 65 78 65 90 00 }

    condition:
       $example_90_04_first_rule and $example_90_04_second_rule
}
```

Pattern 90 05 XX YY

The pattern `90 05 XX YY` is another placeholder for a regex-like expression, where `XX` represents **the upper bound on the number of bytes** that must be found, and `YY` represents the length of the regex-like pattern. The main difference with `90 04 XX YY` is that `90 05 XX YY` is case insensitive when dealing with patterns like `90 05 XX 03 61 2D 7A`.

An example of the pattern `90 05 XX YY` is depicted in **Figure 21**:

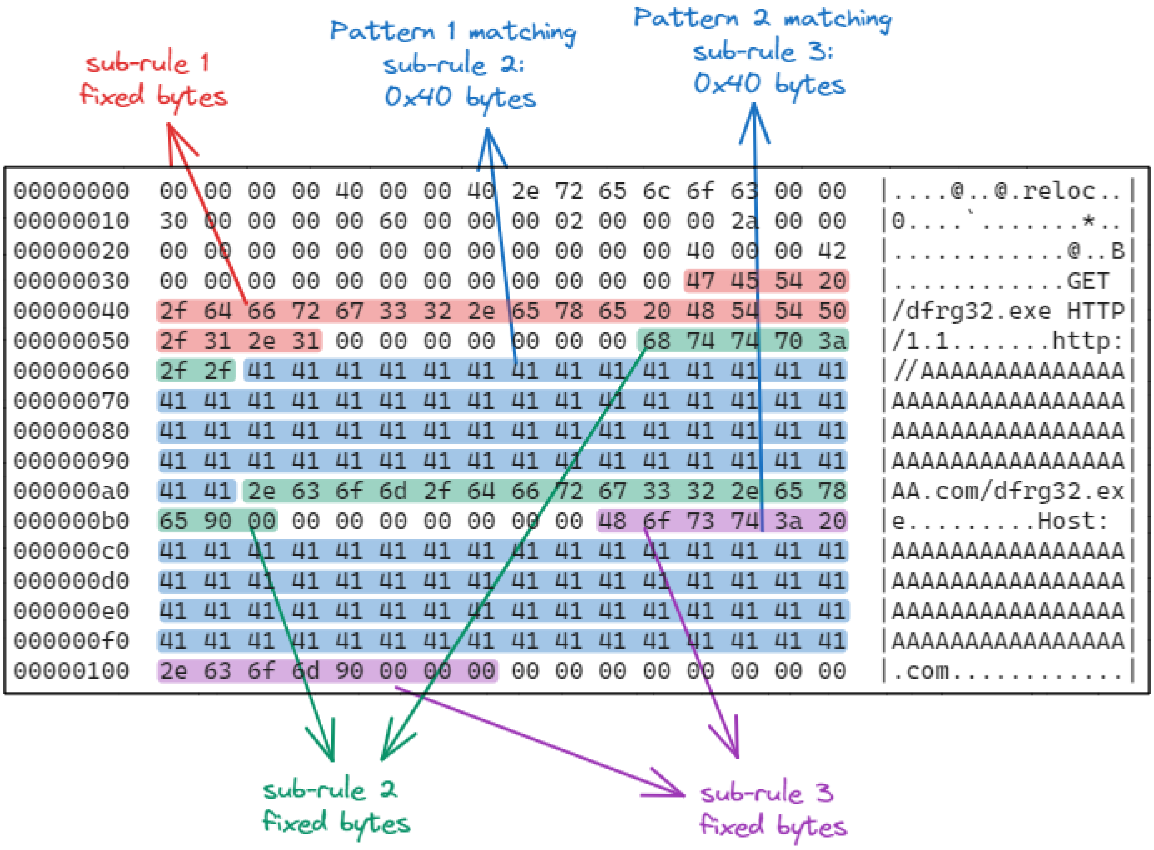


```
00000000  dc e6 2b 01 00 80 5d 04 00 00 e6 ad 01 80 5c 23  |Üæ+...]...æ...\#|
00000010  00 00 e7 ad 01 80 00 00 01 00 04 00 0d 00 88 21  |..ç............!|
00000020  53 74 72 61 74 69 6f 6e 2e 43 43 00 00 01 40 05  |Stration.CC...@.|
00000030  82 5c 00 04 00 78 5f 00 00 1e 00 1e 00 03 00 00  |.\...x_.........|
00000040  0a 00 18 00 47 45 54 20 2f 64 66 72 67 33 32 2e  |....GET /dfrg32.|
00000050  65 78 65 20 48 54 54 50 2f 31 2e 31 0a 00 1f 02  |exe HTTP/1.1....|
00000060  68 74 74 70 3a 2f 2f 90 05 40 03 61 2d 7a 2e 63  |http://..@.a-z.c|
00000070  6f 6d 2f 64 66 72 67 33 32 2e 65 78 65 90 00 0a  |om/dfrg32.exe...|
00000080  00 13 02 48 6f 73 74 3a 20 90 05 40 03 61 2d 7a  |...Host: ..@.a-z|
00000090  2e 63 6f 6d 90 00 00 00 5d 04 00 00 e7 ad 01 80  |.com....]...ç...|
```

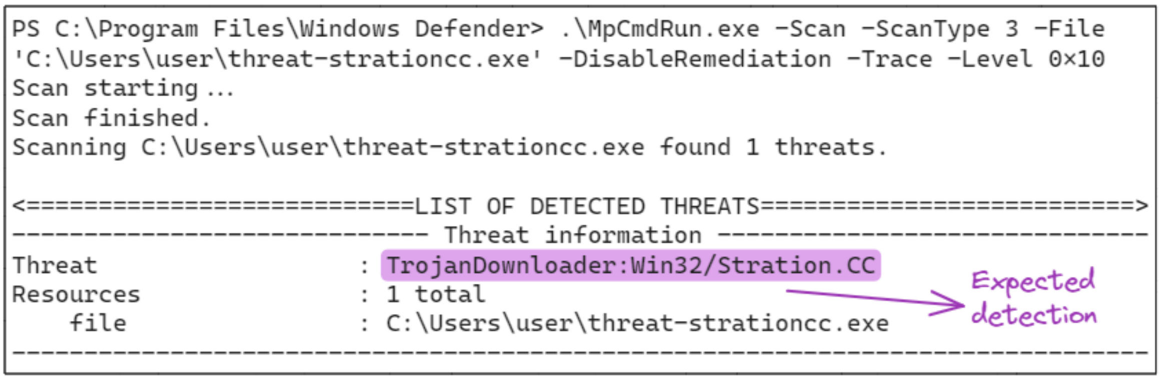
🟩 ui8SubRuleWeightLow 🟧 ui8SubRuleWeightHigh 🟨 ui8SubRuleSize
⬜ ui8CodeUnknown 🟥 pbSubRuleBytesToMatch

In **Figure 21**, the pattern is highlighted in blue within a SIGNATURE_TYPE_PEHSTR_EXT signature for the threat Stration.CC.

Figure 22 shows a PE containing the bytes to trigger the detection:



```
00000000  00 00 00 00 40 00 00 40 2e 72 65 6c 6f 63 00 00  |....@..@.reloc..|
00000010  30 00 00 00 00 60 00 00 00 02 00 00 00 2a 00 00  |0....`.......*..|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42  |............@..B|
00000030  00 00 00 00 00 00 00 00 00 00 00 00 47 45 54 20  |............GET |
00000040  2f 64 66 72 67 33 32 2e 65 78 65 20 48 54 54 50  |/dfrg32.exe HTTP|
00000050  2f 31 2e 31 00 00 00 00 00 00 00 68 74 74 70 3a  |/1.1.......http:|
00000060  2f 2f 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |//AAAAAAAAAAAAAA|
00000070  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000080  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000090  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
000000a0  41 41 2e 63 6f 6d 2f 64 66 72 67 33 32 2e 65 78  |AA.com/dfrg32.ex|
000000b0  65 90 00 00 00 00 00 00 00 00 48 6f 73 74 3a 20  |e.........Host: |
000000c0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
000000d0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
000000e0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
000000f0  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000100  2e 63 6f 6d 90 00 00 00 00 00 00 00 00 00 00 00  |.com............|
```



```
PS C:\Program Files\Windows Defender> .\MpCmdRun.exe -Scan -ScanType 3 -File
'C:\Users\user\threat-strationcc.exe' -DisableRemediation -Trace -Level 0x10
Scan starting...
Scan finished.
Scanning C:\Users\user\threat-strationcc.exe found 1 threats.

<===========================LIST OF DETECTED THREATS==========================>
----------------------------- Threat information ------------------------------
Threat                  : TrojanDownloader:Win32/Stration.CC
Resources               : 1 total
    file                : C:\Users\user\threat-strationcc.exe
-------------------------------------------------------------------------------
```

As can be seen from figure 22, the expected pattern for the detection is properly reproduced, with uppercase characters from `41` to `5A`.

The Yara counterpart for the pattern can be defined as follows:

```
rule Pattern_90_05_example
{
    strings:
       $example_90_05 = "http://[a-zA-Z]{0,64}\\.com/dfrg32\\.exe"

    condition:
       $example_90_05
}
```

Despite the incorporation of wildcards and complex patterns to enhance flexibility and strength, the fact remains that rules based solely on byte and string matching are relatively easy to bypass. Wildcards were introduced to address this limitation by allowing variations in the byte sequences that the rules match. However, these rules can still be evaded by motivated threat actors who employ techniques to modify or obfuscate byte patterns in ways that avoid detection. Therefore, while wildcard patterns improve the robustness of signature-based detection, they are not a foolproof solution against all evasion techniques.

Conclusion

In this analysis we investigated how MDA manages its signatures, with a focus on PEHSTR and PEHSTR_EXT. Armed with this knowledge, in the context of adversary emulation, we can now write an artifact that triggers a specific detection or we could repurpose a detected artifact to evade a particular signature. Of course a proper emulation goes beyond a pattern based detection, but nevertheless it's an interesting case study to showcase the importance of understanding the internals of security solutions.