Jun 28

An unexpected journey into Microsoft Defender's signature World

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

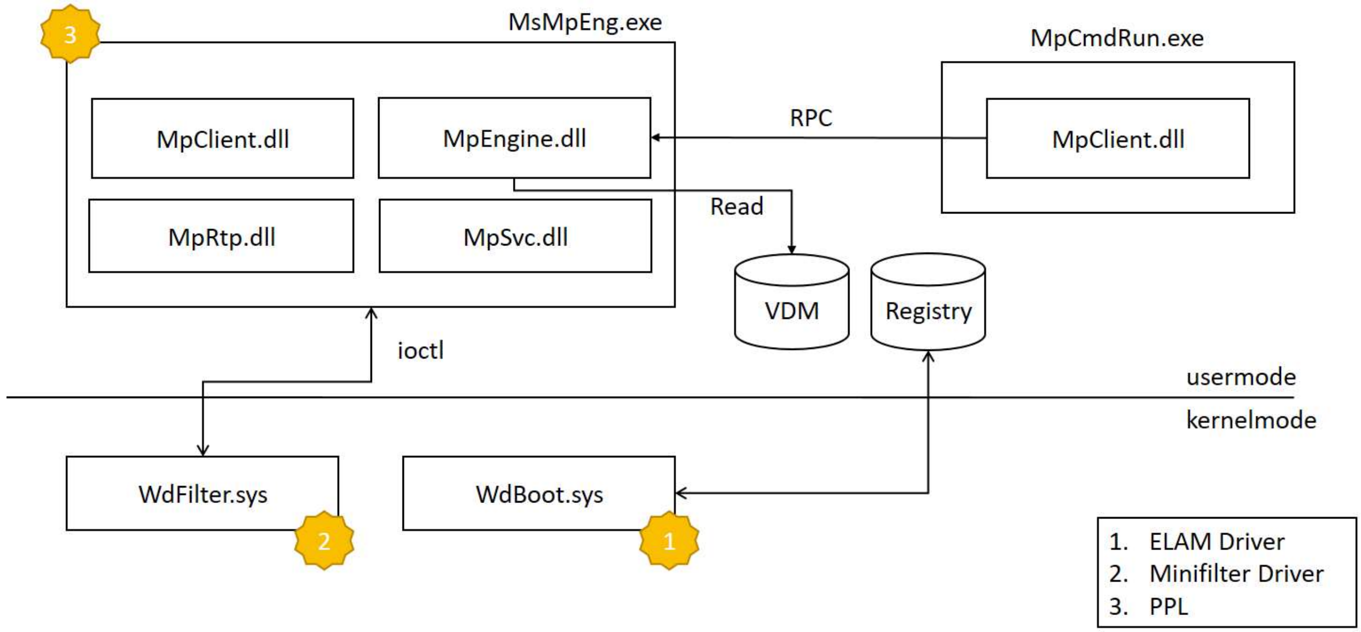


Figure 1. The Microsoft Defender Antivirus (MDA) Architecture

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>\`.

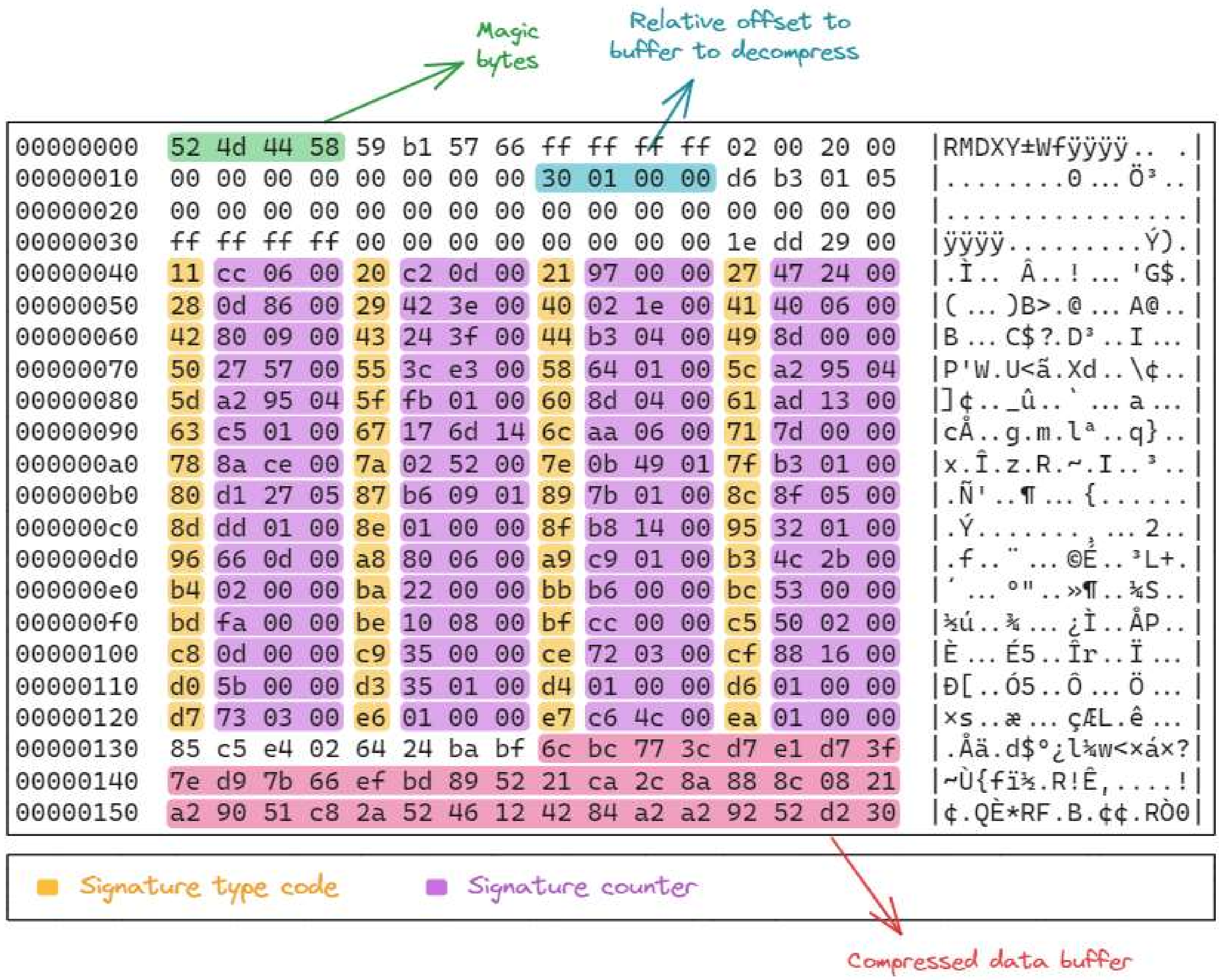


Figure 2. The `RMDX` resource

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).

```
137 pehstr_record_cnt = ESTIMATED_RECORDS(0x61);
138 pehstr_ext_record_cnt = ESTIMATED_RECORDS(0x78);
139 pehstr_ext2_record_cnt = ESTIMATED_RECORDS(0x85);
140 if ( pehstr_ext_record_cnt + pehstr_record_cnt < pehstr_record_cnt
141   || (pe_hstr_total_cnt = pehstr_record_cnt + pehstr_ext_record_cnt + pehstr_ext2_record_cnt,
142       (unsigned int)pe_hstr_total_cnt < pehstr_ext_record_cnt + pehstr_record_cnt) )
143 {
144   v0 = 32780;
145   goto out_1;
146 }
147 g_pe_hstr_total_cnt = pehstr_record_cnt + pehstr_ext_record_cnt + pehstr_ext2_record_cnt;
148 g_p_pehstr_total = (__int64)calloc(pe_hstr_total_cnt, 0x14ui64);
149 if ( !g_p_pehstr_total )
150   goto out;
151 byte_[illegible] = 0;
152 curr_handler.pfn_push = (UINT64)hstr_push;
153 curr_handler.hstr_type = 0x61;
154 curr_handler.pfn_pushend = (__int64 (__fastcall *)())hstr_pushend_common;
155 p_gHstrSigs = (char *)&g_HstrSigs;
156 v0 = regcntl(&curr_handler, 0x30ui64, 0xC);
157 if ( v0 )
158   goto out_1;
159 curr_handler.pfn_push = (UINT64)hstr_push_ext;
160 curr_handler.hstr_type = 0x78;
161 curr_handler.pfn_pushend = (__int64 (__fastcall *)())hstr_pushend_common;
162 p_gHstrSigs = (char *)&g_HstrSigs;
163 v0 = regcntl(&curr_handler, 0x30ui64, 0xC);
164 if ( v0 )
165   goto out_1;
166 curr_handler.pfn_push = (UINT64)hstr_push_ext2;
167 curr_handler.hstr_type = 0x85;
168 curr_handler.pfn_pushend = (__int64 (__fastcall *)())hstr_pushend_common;
169 p_gHstrSigs = (char *)&g_HstrSigs;
170 v0 = regcntl(&curr_handler, 0x30ui64, 0xC);
171 if ( v0 )
172   goto out_1;
```

Figure 3. The decompiled version of the function `mpengine!pattsearch_init`

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.

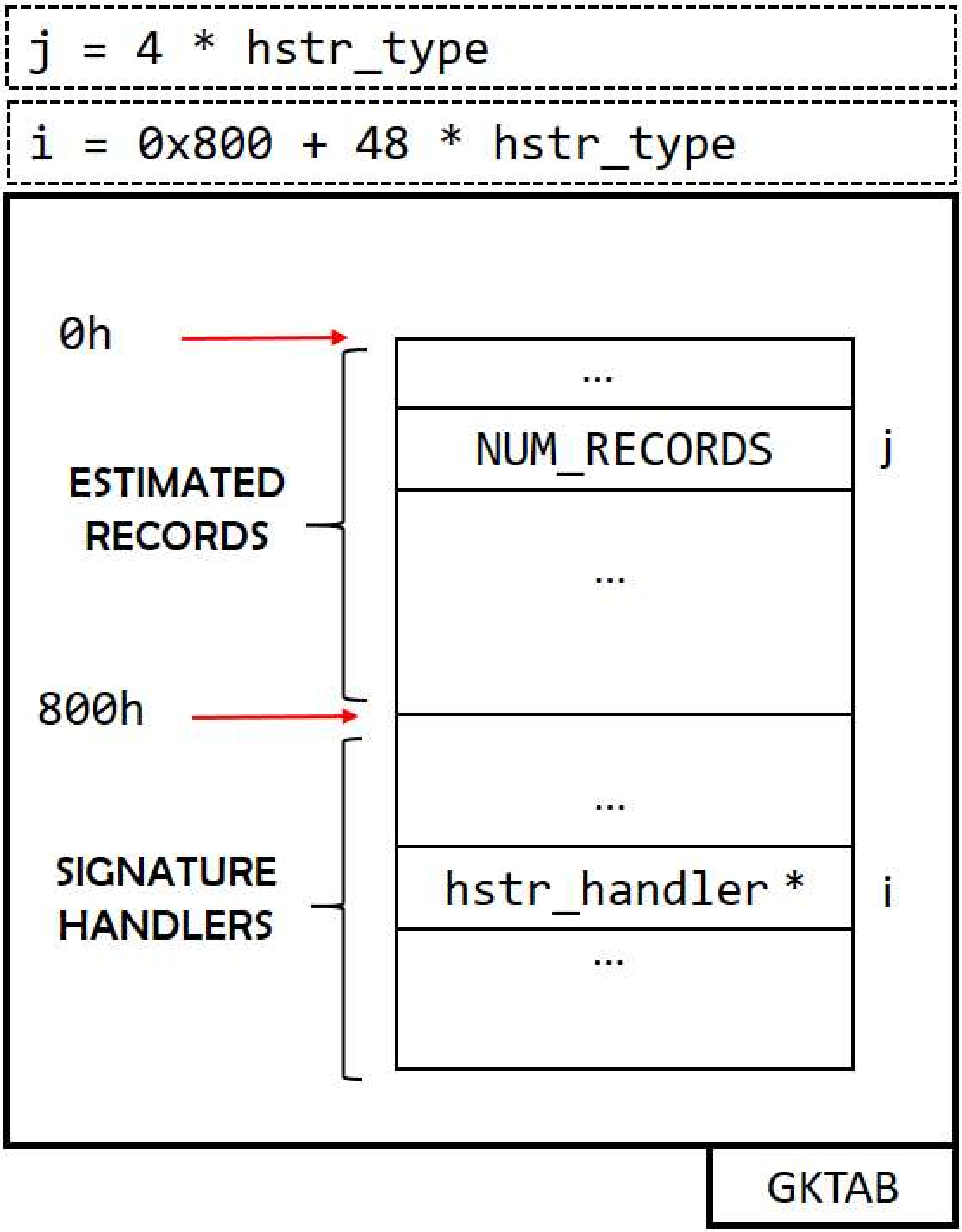


Figure 4. Memory layout of record numbers and signature handlers

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).

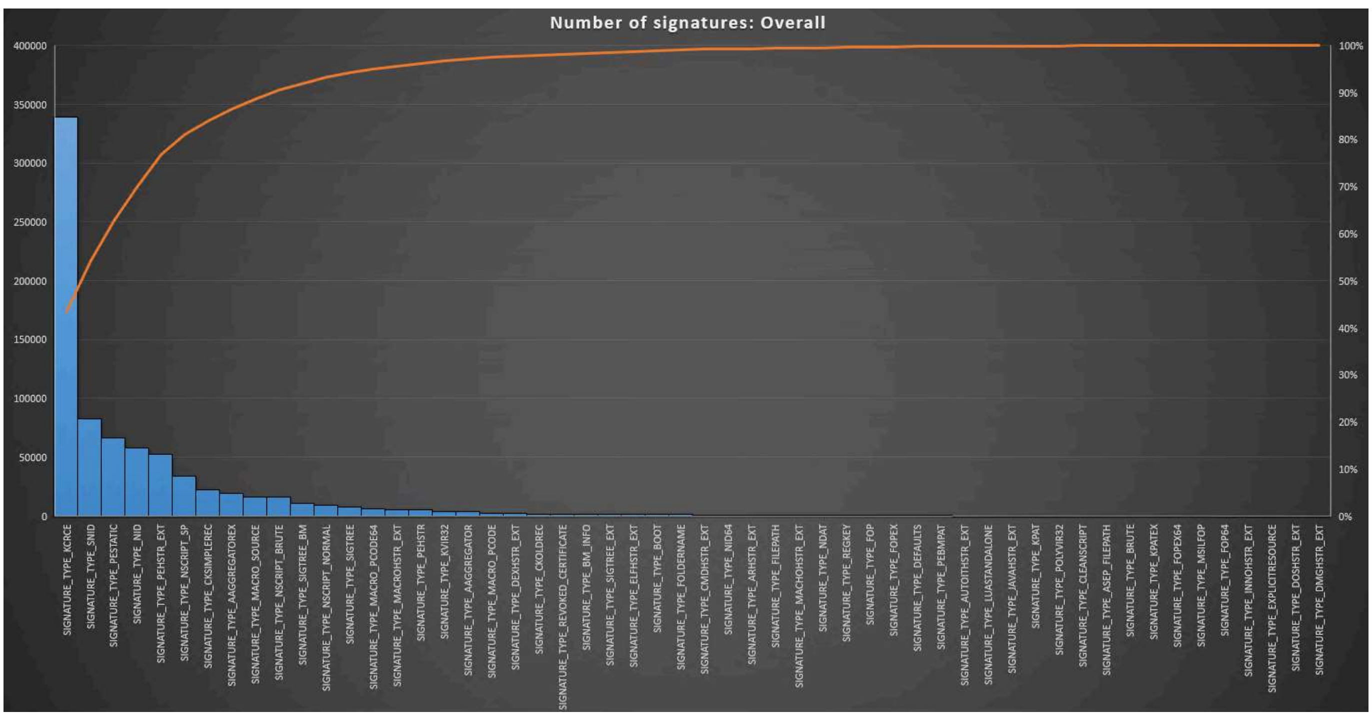
pefile_module and the hstr_internal_search

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

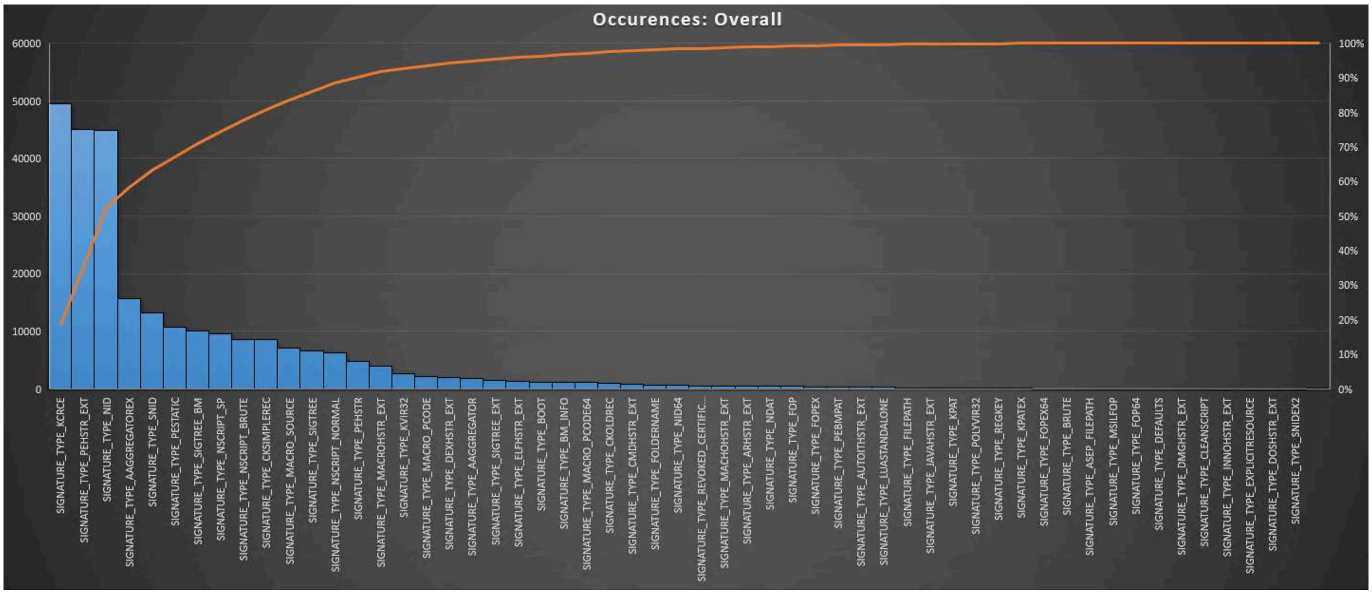
Before delving into the signature type analysis we report few stats on the different signature types that are present into the VDM file. To have an idea of signature distribution we wrote a simple python script that parses the VDM and counts the number of signatures by types.



`Signature count`

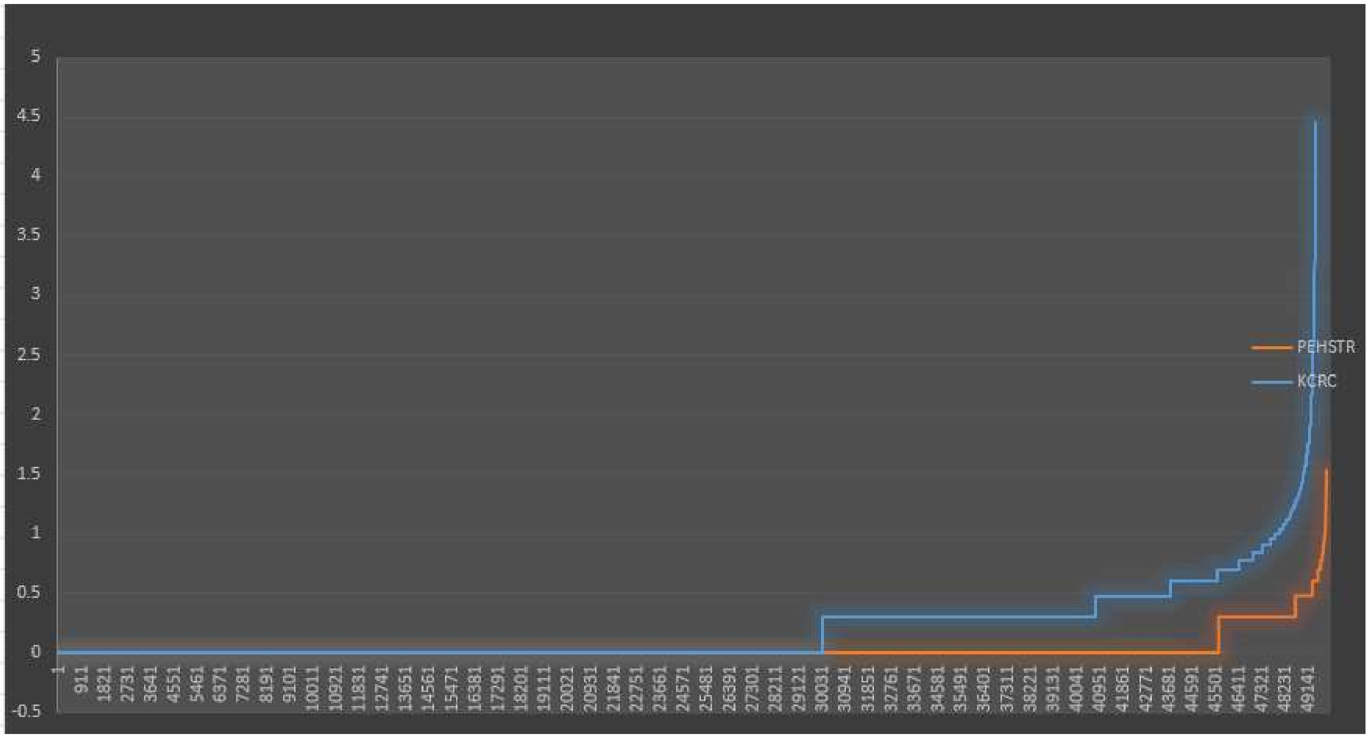
We noticed that the `SIGNATURE_TYPE_KCRCE` is by far the most frequent with more than `338k` signatures. Indeed, even by stacking all the `HSTR` family signature types together we don't even reach 100k signatures. Far below the `KCRCE`.

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.



Signature Occurrences

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.



PEHSTR vs KCRCE signature distribution

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.

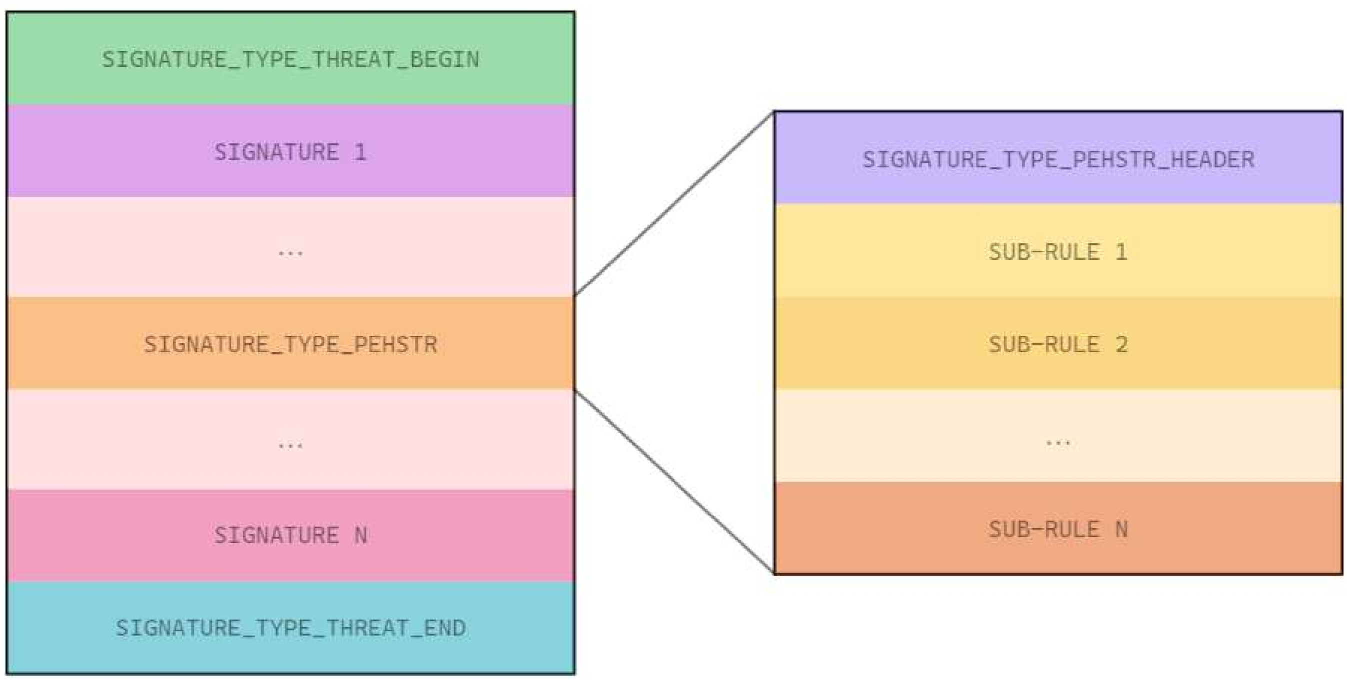


Figure 5. Threat layout inside signature files.

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;

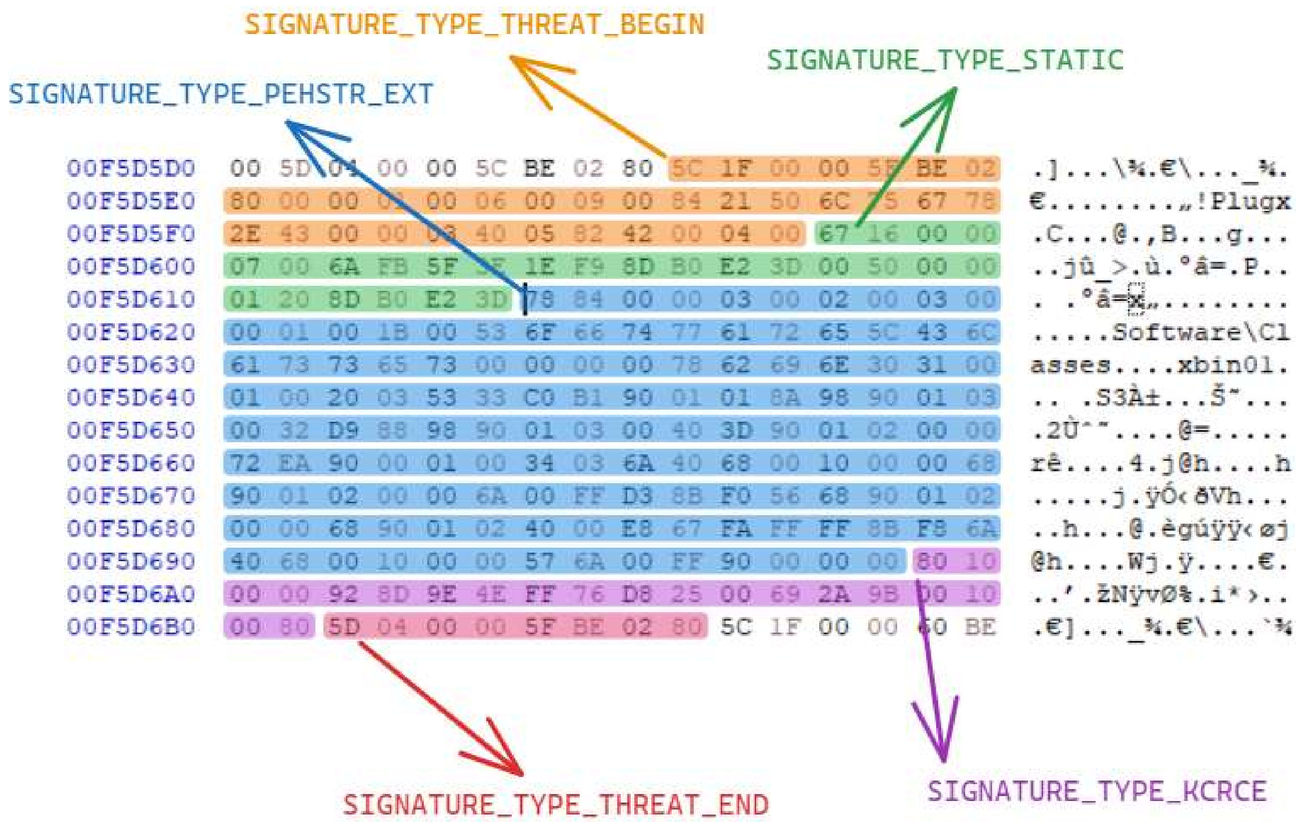


Figure 6: Plugx.C signatures example.

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**:

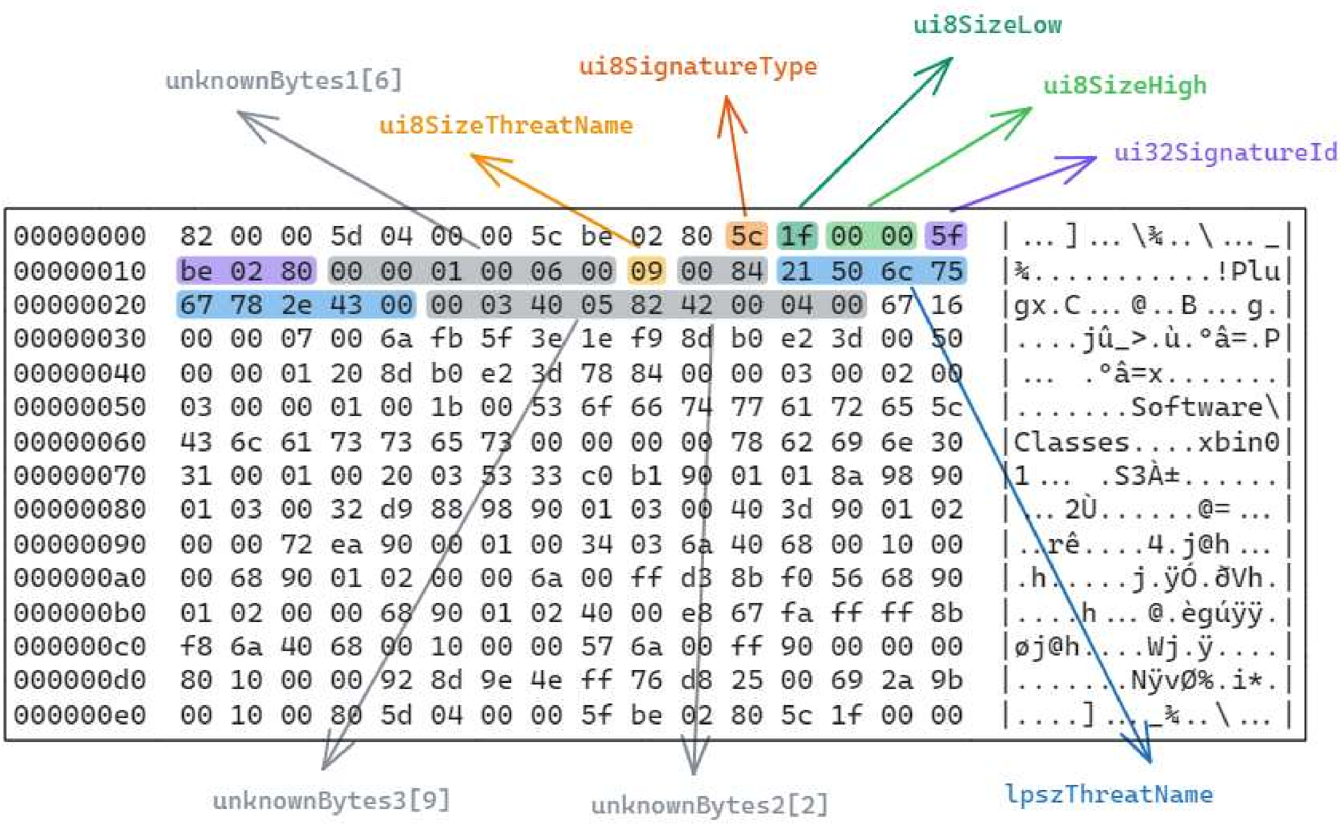


Figure 7: Plugx.C `SIGNATURE_TYPE_THREAT_BEGIN` example.

`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`.

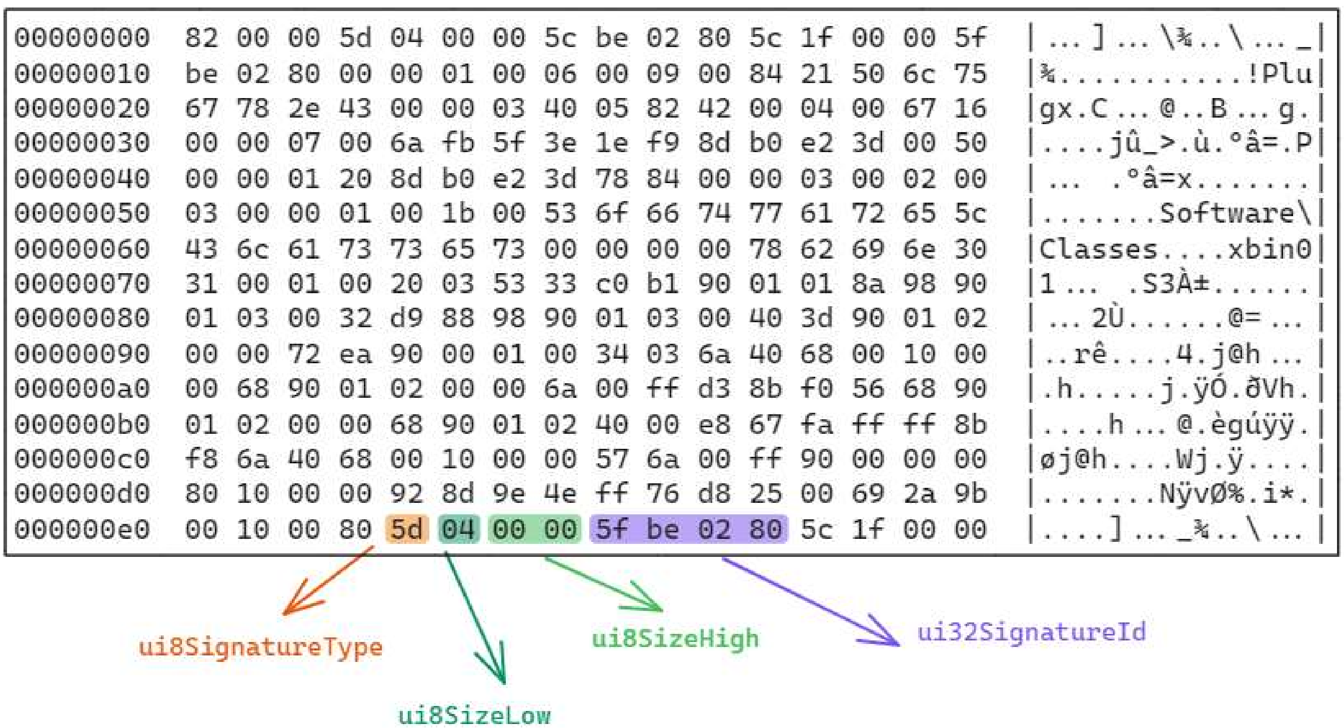


Figure 8: Plugx.C `SIGNATURE_TYPE_THREAT_END` breakdown.

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.

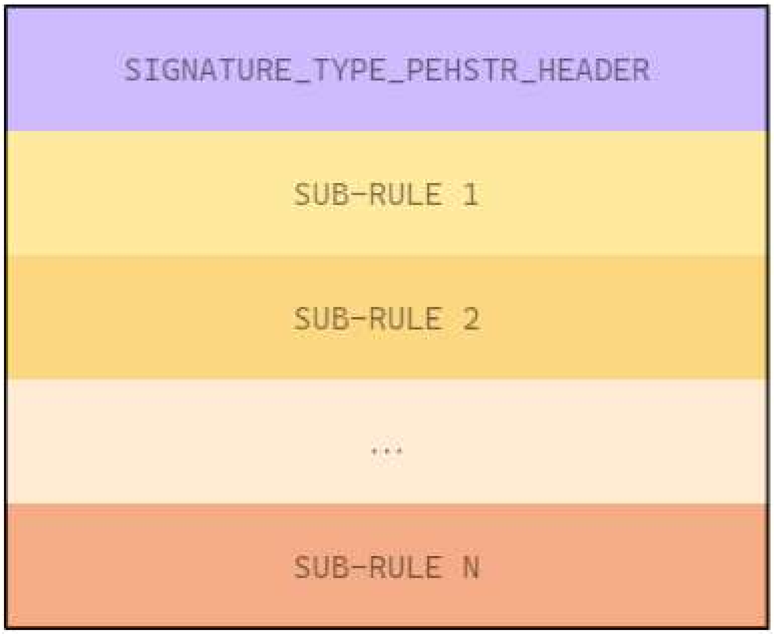


Figure 9: `SIGNATURE_TYPE_PEHSTR` breakdown breakdown.

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`:

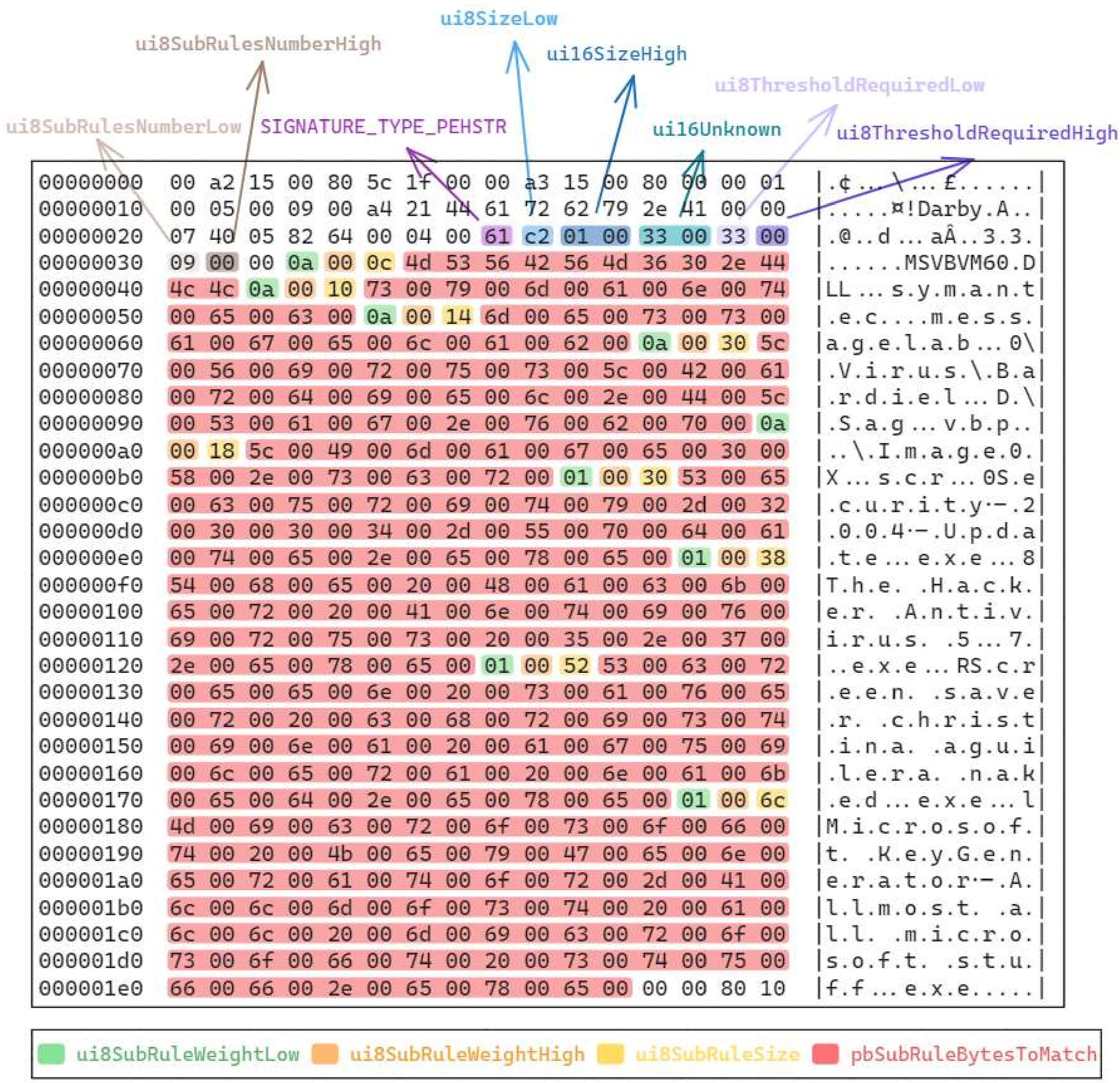


Figure 10: Darby.A `SIGNATURE_TYPE_PEHSTR` breakdown.

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:

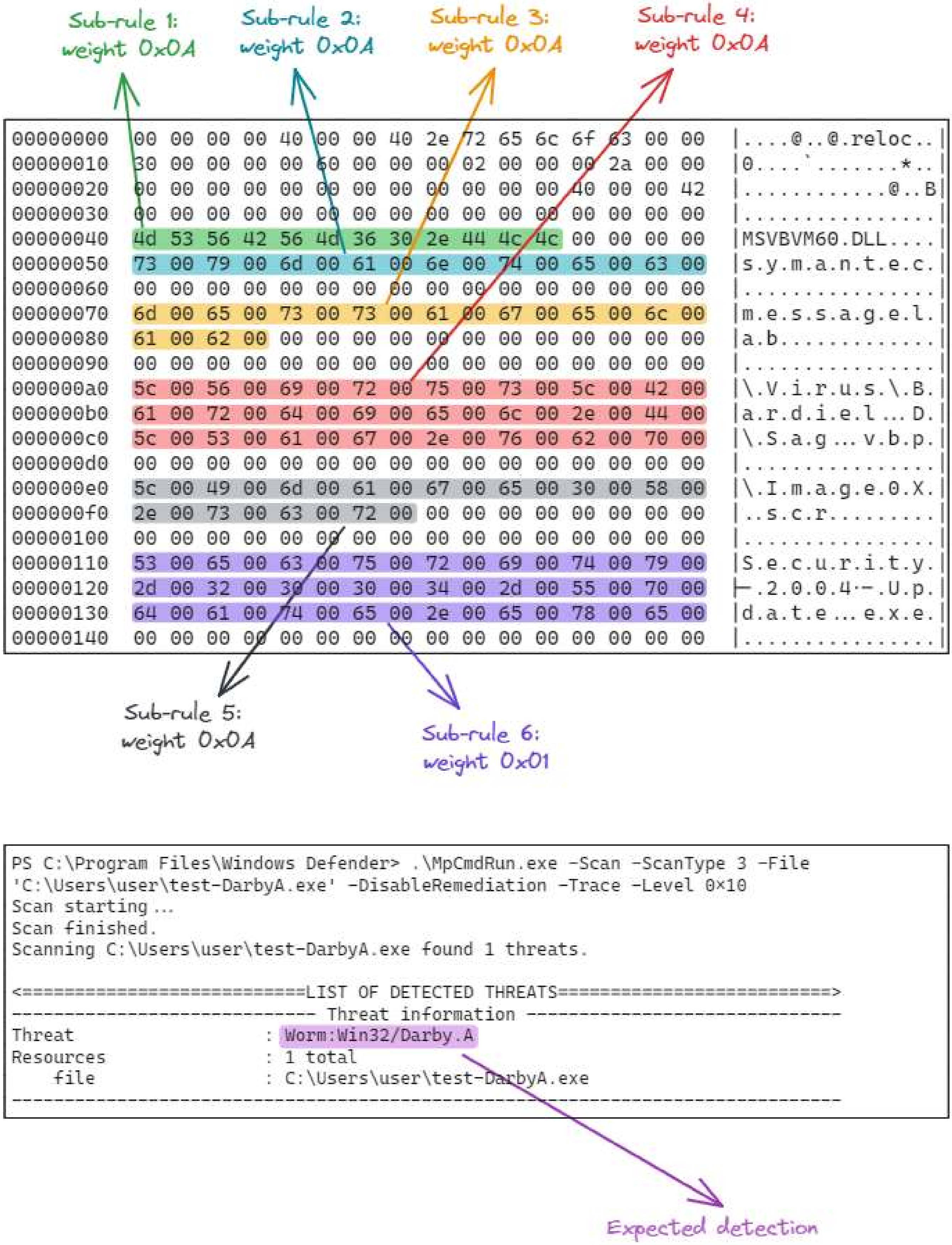


Figure 11: `SIGNATURE_TYPE_PEHSTR` detection example.

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:

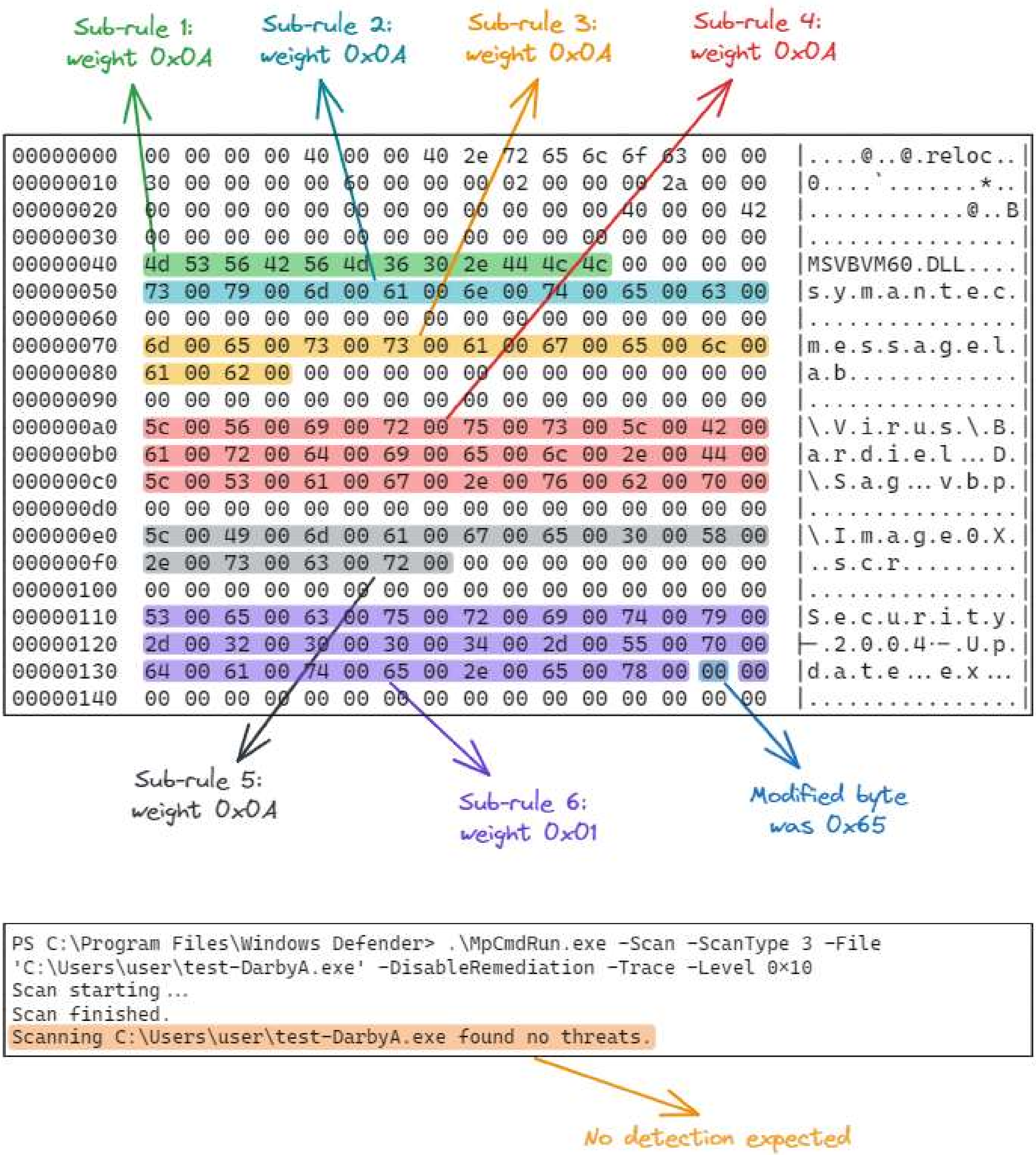


Figure 12: `SIGNATURE_TYPE_PEHSTR` no detection example.

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

All wildcard rules start with a byte `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:

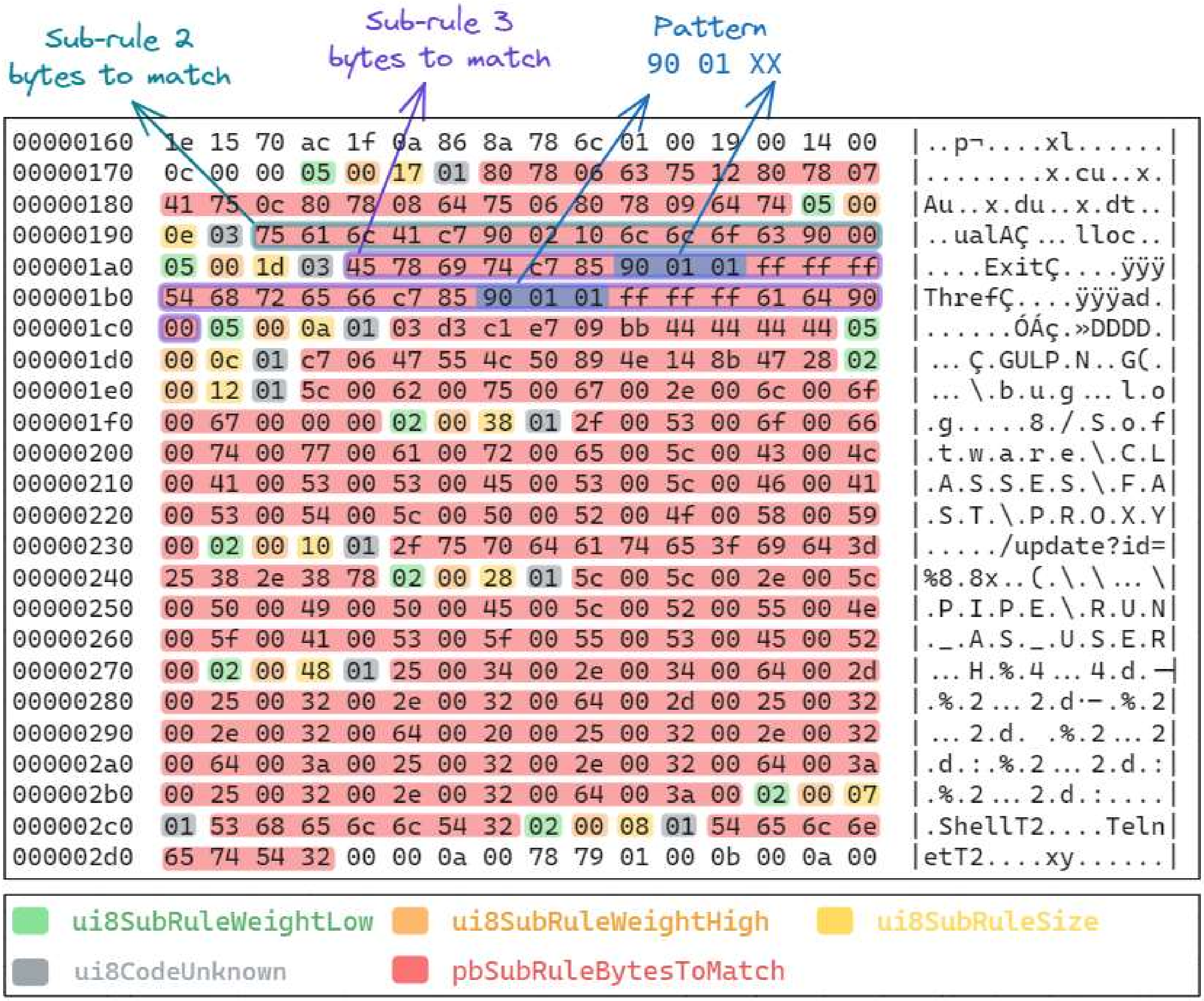


Figure 13: Example of pattern `90 01 XX`.

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**.

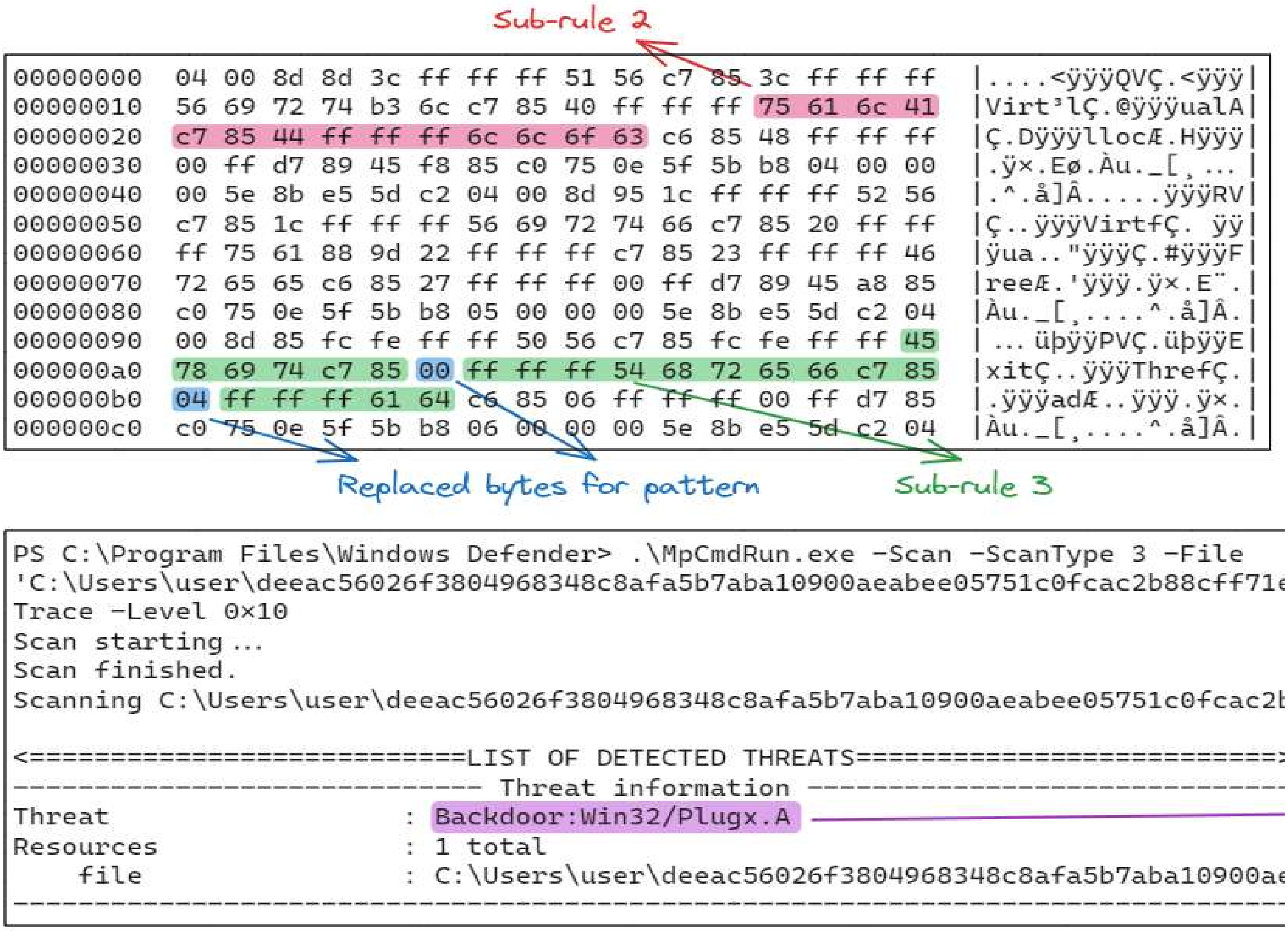


Figure 14: PE containing `Plugx.A` sub-rules with `90 01 XX` pattern.

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.

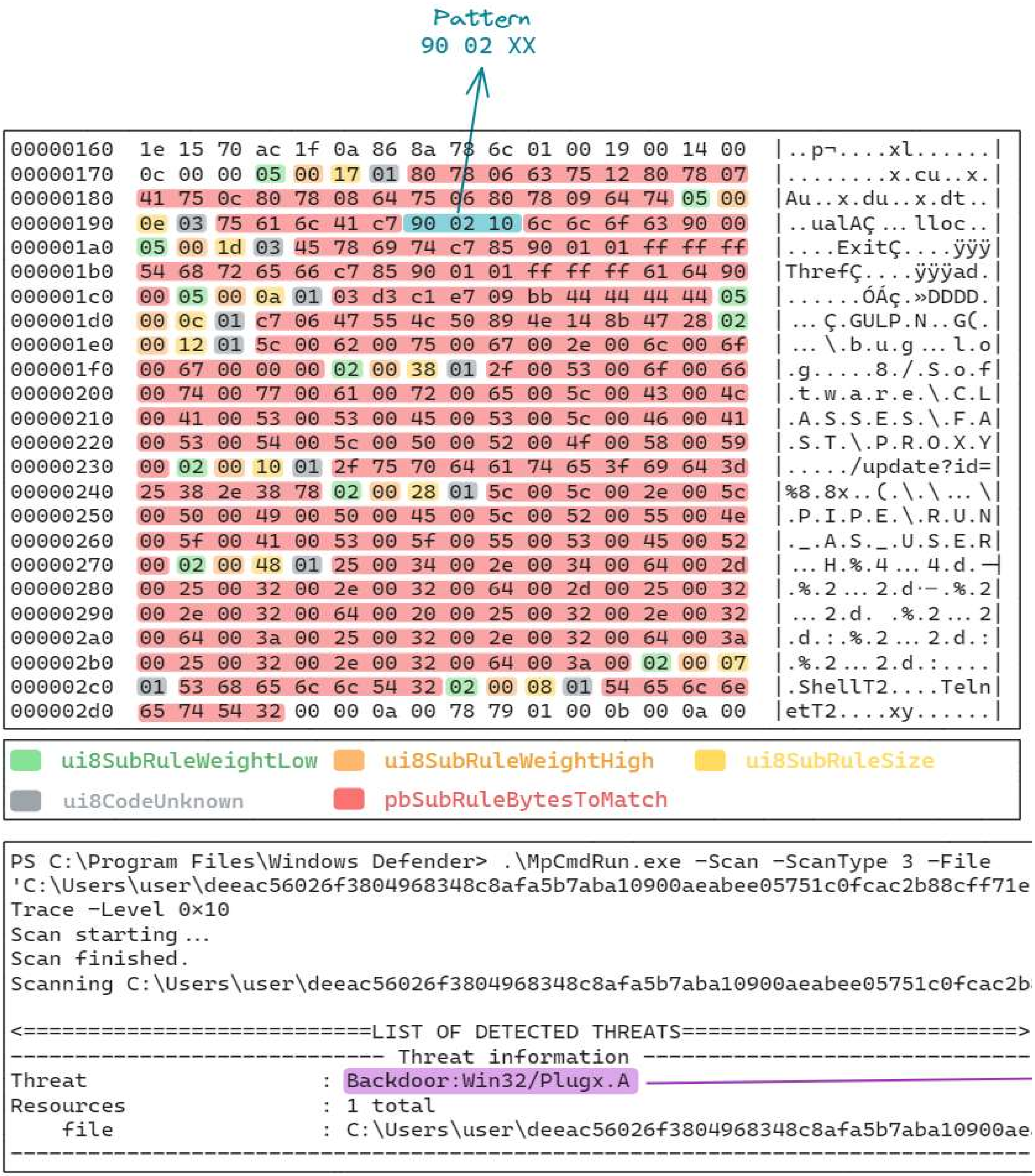


Figure 15: Example of pattern `90 02 XX` which triggers the detection as PlugX.A

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**:

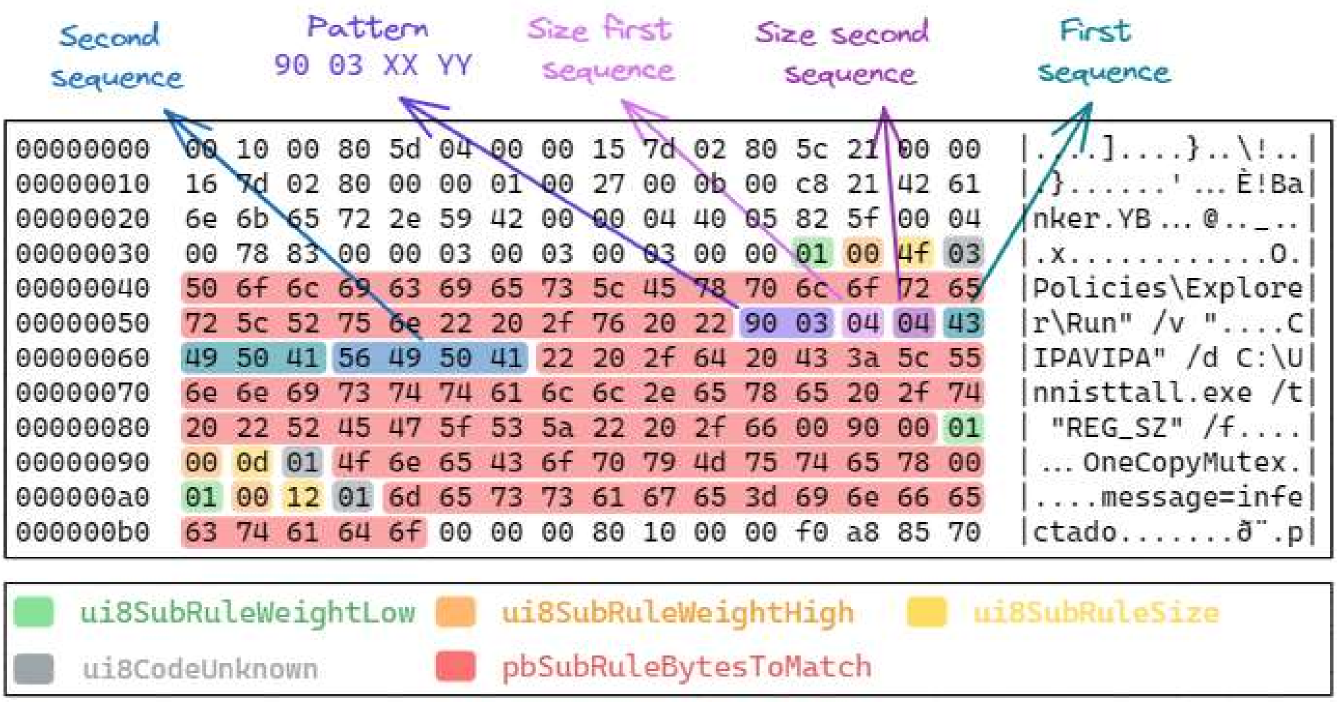


Figure 16: Example of pattern `90 03 XX YY` with string sequences.

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**:

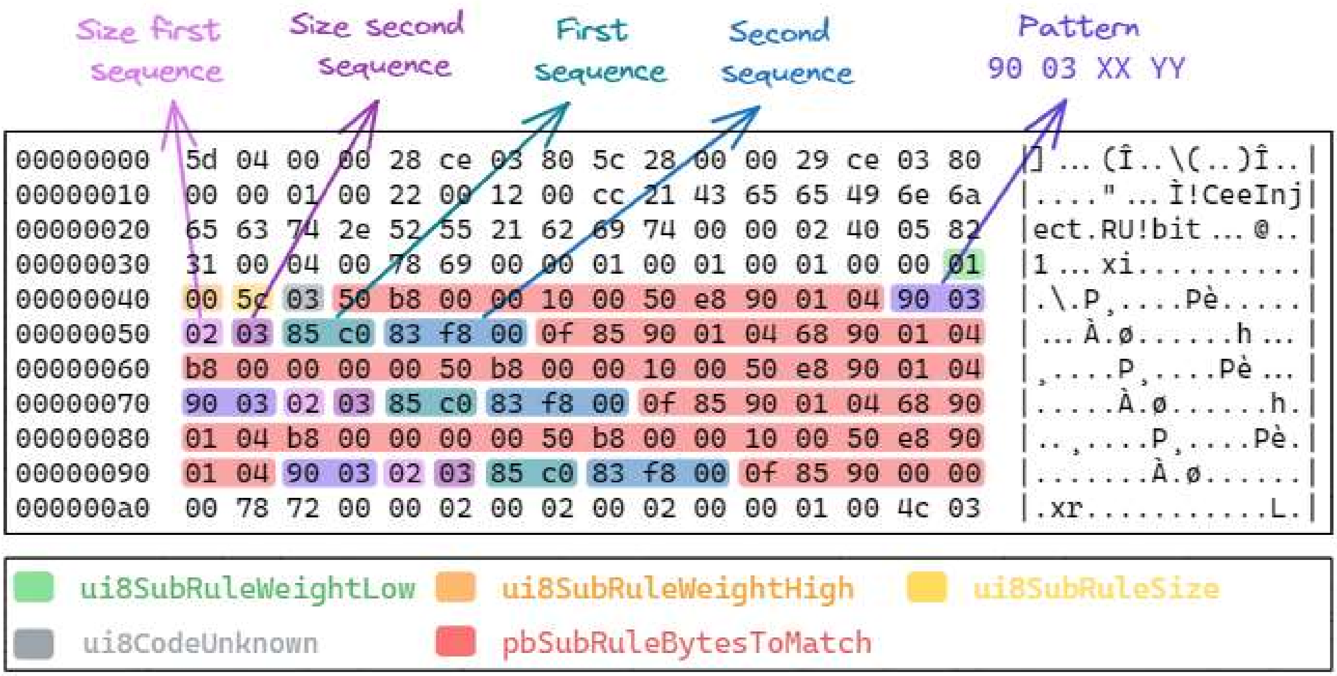


Figure 17: Example of pattern `90 03 XX YY` with byte sequences.

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:

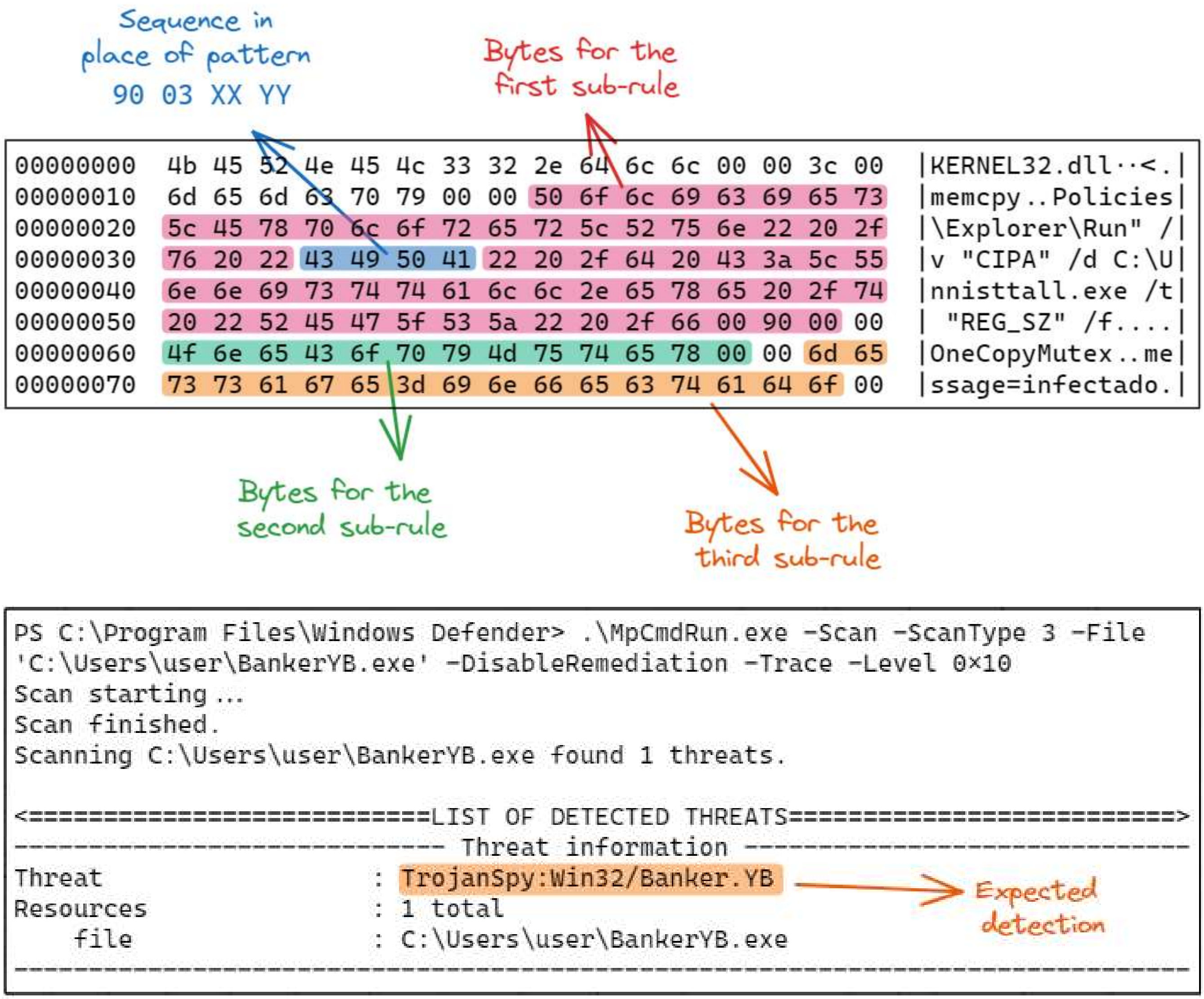


Figure 18: Example of PE containing bytes for the pattern `90 03 XX YY` which triggers the detection of threat `Banker.YB`.

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**:

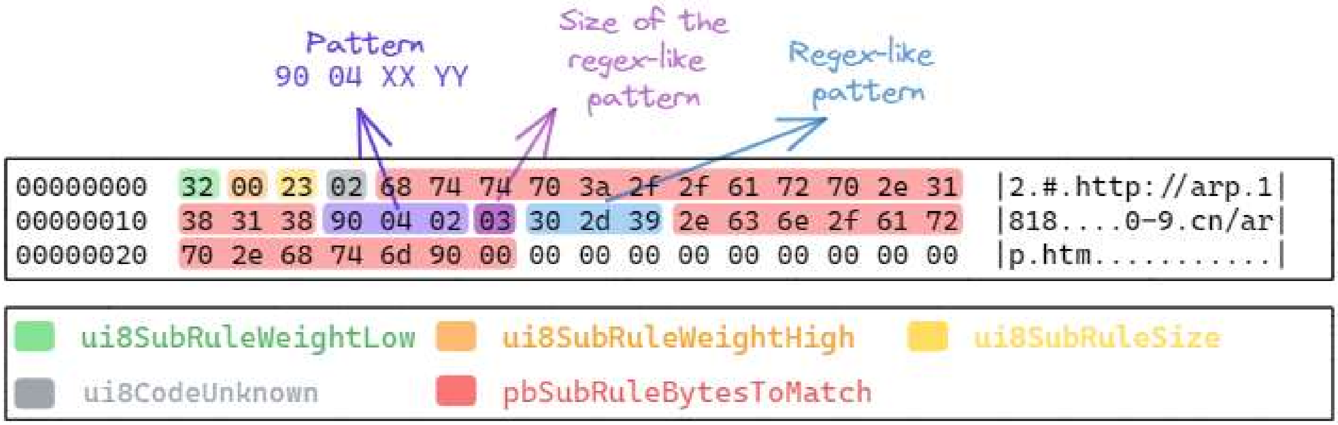


Figure 19: Example of pattern `90 04 XX YY`.

More complex patterns can contain the exact characters to match:

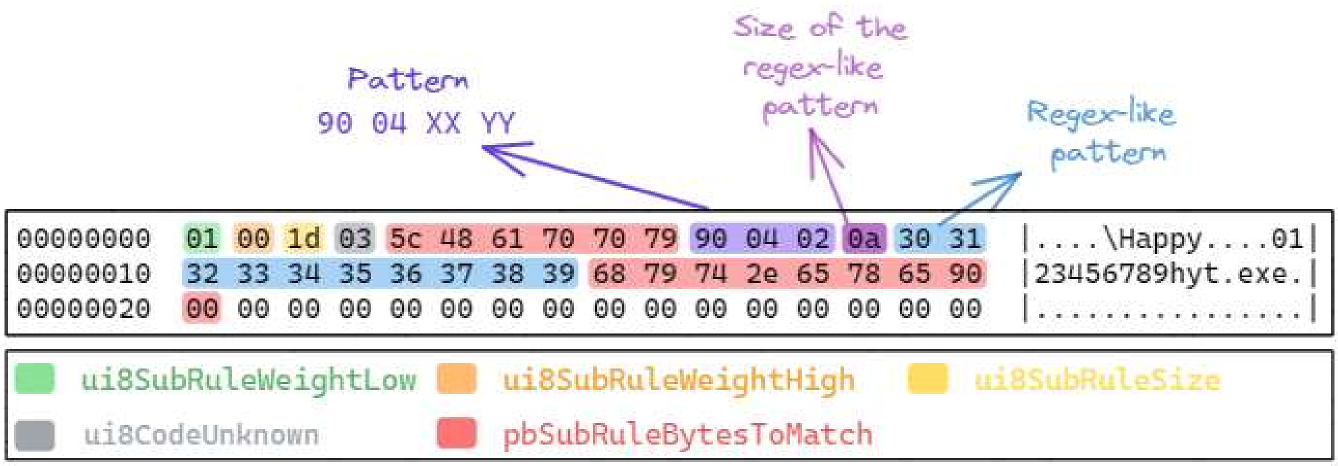


Figure 20: Example of longer pattern `90 04 XX YY`.

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**:

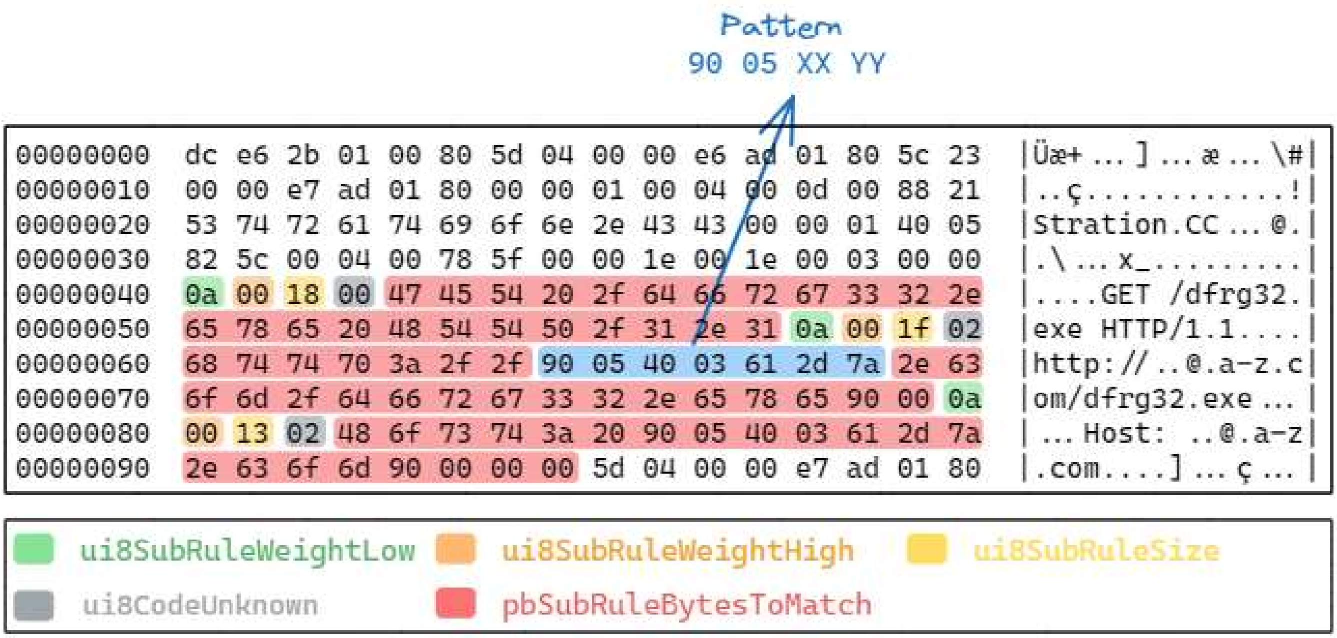


Figure 21: Example of pattern `90 05 XX YY`.

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:

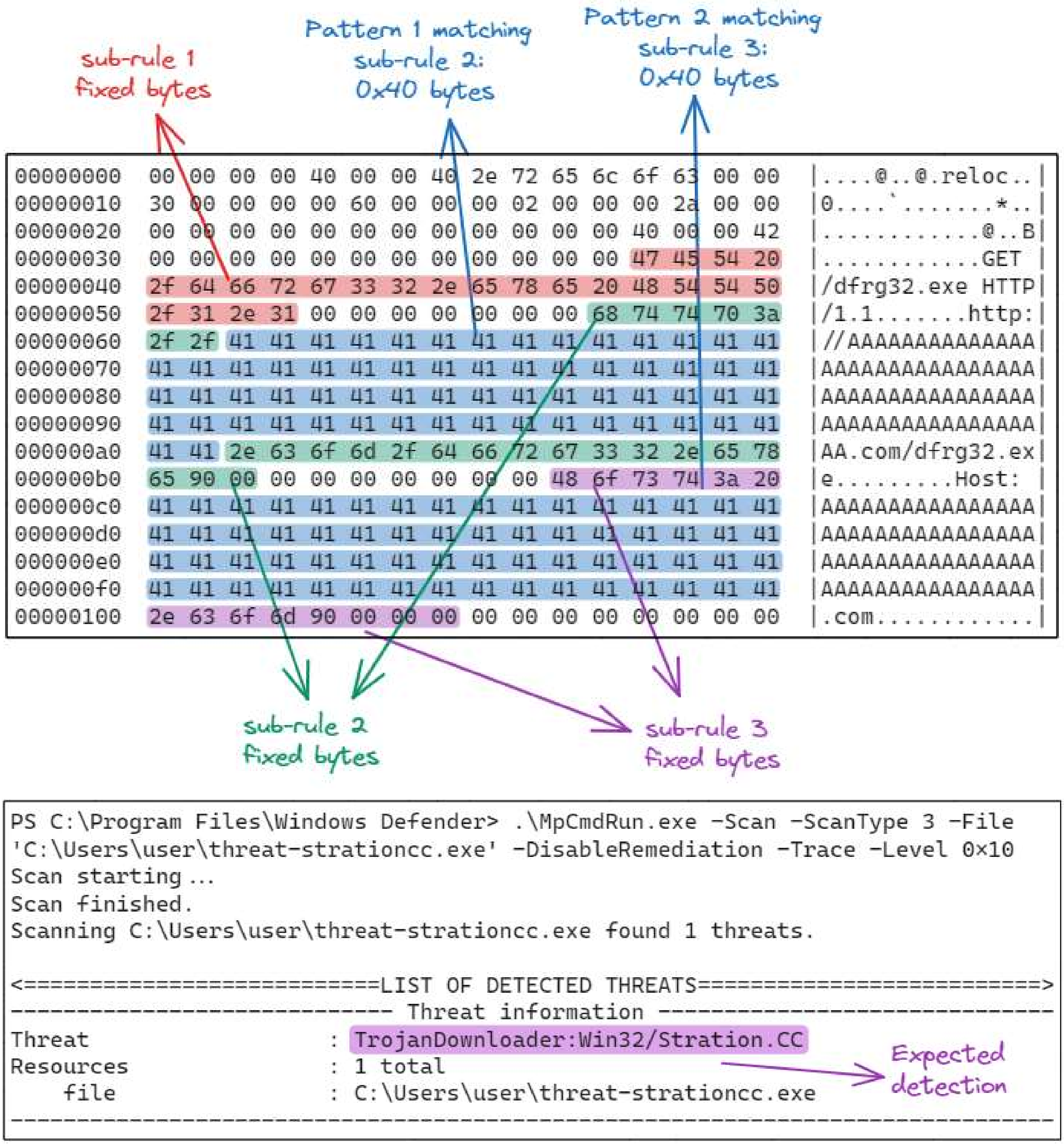


Figure 22: `Stration.CC` Portable Executable which triggers the detection as `Stration.CC`

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.

Contact