



Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	(Ελληνικά) Δημιουργία και Αλλαγή prefetch αρχείων για Windows 10 (Αγγλικά) Manipulating and generating Windows 10 Prefetch files
Όνοματεπώνυμο Φοιτητή	Βασίλης Βουβούτσης
Πατρώνυμο	Χρήστος
Αριθμός Μητρώου	ΜΠΣΠ/17009
Επιβλέπων	Πατσάκης Κωνσταντίνος

Ημερομηνία Παράδοσης 06/2019

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)

Πατσάκης Κωνσταντίνος
Επίκουρος Καθηγητής

(υπογραφή)

Αλέπης Ευθύμιος
Επίκουρος Καθηγητής

(υπογραφή)

Τσιχριντζής Γεώργιος
Καθηγητής

Table of Contents

1 Abstract.....	4
1.1 English.....	4
1.2 Greek.....	4
2 Introduction.....	5
2.1 Forensics Value.....	5
2.2 Parameters.....	6
2.3 Prefetch files.....	7
2.4 Windows 10 prefetch files.....	7
2.4.1 MAM file format.....	7
2.4.2 File Header.....	7
2.5 Compression Engines.....	8
2.5.1 LZXPRESS Huffman.....	8
2.5.2 Compression API.....	8
2.5.3 Layout.ini.....	9
3 Standby Memory.....	9
4 Compress Procedure.....	10
4.1 Decompress.....	10
4.2 Using the compression API.....	10
4.3 Generating the Header.....	11
4.4 Manipulating the prefetch file.....	11
5 Scenarios.....	14
5.1 Fake Entries.....	14
5.2 Storing Information.....	15
5.3 Loading into Memory.....	15
6 Results.....	17
7 Python Source Code[18].....	18
7.1 Python Imports.....	18
7.2 Enumeration Classes.....	18
7.3 Helper Function.....	18
7.4 Compression Method Function.....	18
7.5 Header Calculation Function.....	19
7.6 SSCA 2008 Hash Function.....	20
8 Sample Prefetch File Structure Tables.....	21
8.1 File header.....	21
8.2 Compressed Data Block.....	21
8.3 Uncompressed Data Block.....	21
8.3.1 File Header.....	21
8.3.2 File information.....	21
8.3.3 File metrics array entry.....	22
8.4 Trace chains array.....	22

1 Abstract

1.1 English

The prefetch file format is not officially documented by Microsoft and has been understood through reverse engineering, and trial-and-error. Without even intending to do so, prefetch files can sometimes answer the vital questions of computer forensic analysis: who, what, when, where, why, and sometimes even how. Even if they are designed to speed up the system's disk read times, can also be used for a more efficient intrusion disguise or to increase the operating system's attack surface. When a Windows system boots, components of many files need to be read into memory and processed. Since windows 10, prefetch files are no more clear text, but instead are compressed. But we now know that an attacker can re-compress prefetch files and manipulate them by hiding or adding entries to the files.

1.2 Greek

Η μορφή αρχείου prefetch δεν τεκμηριώνεται επισήμως από τη Microsoft και έχει γίνει κατανοητή με αντίστροφη μηχανική ή δοκιμασία και σφάλμα. Χωρίς κάποια πρόθεση, τα αρχεία prefetch, μπορούν μερικές φορές να απαντήσουν στα ζωτικά ερωτήματα της εγκληματικής ανάλυσης: ποιος, τι, πότε, πού, γιατί, και μερικές φορές ακόμη και πώς. Ακόμη και αν έχουν σχεδιαστεί για να επιταχύνουν τους χρόνους ανάγνωσης του δίσκου του συστήματος, μπορούν επίσης να χρησιμοποιηθούν για μια πιο αποτελεσματική μεταμφίεση εισβολής ή για την αύξηση της επιφάνειας επίθεσης του λειτουργικού συστήματος. Όταν εκκινείται ένα σύστημα Windows, τα στοιχεία πολλών αρχείων πρέπει να φορτωθούν στη μνήμη και να υποστούν επεξεργασία. Δεδομένου ότι από τα Windows 10, τα αρχεία prefetch δεν είναι πλέον καθαρό κείμενο, αλλά αντί αυτού είναι συμπιεσμένα. Τώρα όμως γνωρίζουμε ότι ένας εισβολέας μπορεί να συμπιέσει ξανά αρχεία prefetch και να τα χειριστεί κρύβοντας ή προσθέτοντας καταχωρήσεις στα αρχεία.

2 Introduction

The Prefetcher is a Microsoft Windows component that was introduced in Windows XP. [1] Is part of Windows' Memory Manager System that can accelerate Windows' initialization process and reduce the applications start up times. This is achieved with the loading and temporary storage of all the files and libraries, that are required by an application to execute, into the RAM during a program's start up, achieving this way reduced Hard Disk read operations. This component is covered by the USA Patent. [2] Since Windows Vista, the Prefetcher has been extended to SuperFetch and ReadyBoost.

When the Windows Operating System starts, multiple file components must be loaded into the RAM in order to be processed. In most of the times, multiple parts of the same file (e.g. Registry hives) are loaded into the memory in different times. As a result, a significant amount of time is used by jumping, multiple times, from one file to another, although a single file system access would be much more effective. Prefetcher works by monitoring witch files are accessed during the system's start up process, including the NTFS' Master File Table, and the generation of a monitor file of this process. Prefetcher will continue monitoring this activity until 30 seconds after the start up of the users environment have passed or 60 seconds after the initialization of all services or 120 seconds after the system's start up. Whichever comes first.

Future boots can then use this information that have been monitored in this activity monitoring file, in order to load the applications data with a more efficient way (i.e. by reproducing disk readings to minimize or eliminate the need to access the same file multiple times, minimizing disk drive's movements).

Application Prefetch works in a similar way, but it is detected when starting an application. Only the first 10 seconds of activity are tracked. [1]

The file itself will contain metadata such as the executable's name, files and directories that the application uses during the first 10 seconds of execution, the prefetch file size, the volume path, the serial number, the execution number, the creation time and the last execution time of the executable [3]. Other than these elements, in a prefetch file, there is a large set of data that contains instructions to load what the program uses most often at startup.

The Task Scheduler is the process that is responsible for analyzing track data collected by the Prefetcher and recording files in the prefetch folder. As a result, Prefetcher will not work properly if Task Scheduler does not start.

To further improve access time, Task Scheduler calls Windows Disk Defragmentation every three days. When the machine is idle, the lists of files and folders reported during the boot process and application launches. [1] The end result is saved in the Layout.ini file in the Prefetch folder and then passed to Windows Disk Defragmentation by rearranging these files into successive locations on the physical hard disk.

2.1 Forensics Value

The forensic value of the contents of this file is immediately obvious. From the file metadata an examiner can identify that cmd.exe was executed, the location, and frequency. These artifacts might answer the "what" and the "where" of an incident. The number of times executed will increment each time the application is run. The timestamp information indicates when the first time the application was executed and when it was last accessed, or executed. This might answer the "when" some activity of interest occurred. Any file that is configured to automatically "autostart" will not register a prefetch file when it is created. If the prefetch file is deleted from the prefetch folder, both the timestamps and the number of times executed will be reset.[4]

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 17 00 00 00 53 43 43 41 11 00 00 00 C6 36 02 00 [...]...SCCA....E6..
00000010 43 00 48 00 52 00 4F 00 4D 00 45 00 2E 00 45 00 C.H.R.O.M.E...E.
00000020 58 00 45 00 00 00 FF FF 00 00 00 00 00 00 00 00 X.E...ÿÿ.....
00000030 50 00 00 00 80 FA FF FF 00 00 00 00 00 00 00 00 P...€úÿÿ.....
00000040 00 00 00 00 00 00 00 00 9E 81 2E 03 BA B1 99 D9 .....ž...°±™Ů
00000050 00 00 00 00 F0 00 00 00 10 01 00 00 F0 22 00 00 ....š.....š"..
00000060 97 1C 00 00 04 7A 01 00 FA B4 00 00 00 2F 02 00 -....z...ú'.../..
00000070 01 00 00 00 C6 07 00 00 0B 00 00 00 01 00 00 00 ....E.....
00000080 99 2E 36 A7 BA A9 CF 01 00 8C 86 47 00 00 00 00 ™.6š°@İ..€+G....
00000090 00 8C 86 47 00 00 00 00 76 05 00 00 07 00 00 00 .€+G....v.....
000000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000F0 00 00 00 00 76 00 00 00 50 00 00 00 00 00 00 00 ....v...P.....
00000100 32 00 00 00 00 02 00 00 2D 62 04 00 00 00 04 00 2.....-b.....
00000110 76 00 00 00 36 00 00 00 31 00 00 00 66 00 00 00 v...6...1...f...
00000120 32 00 00 00 00 02 00 00 EA C4 02 00 00 00 03 00 2.....ëÄ.....
00000130 AC 00 00 00 2E 00 00 00 14 00 00 00 CC 00 00 00 -.....ì....
00000140 35 00 00 00 00 02 00 00 E0 C4 02 00 00 00 03 00 5.....àÄ.....
00000150 DA 00 00 00 06 00 00 00 05 00 00 00 38 01 00 00 Ú.....8....
00000160 35 00 00 00 00 02 00 00 CF C4 02 00 00 00 03 00 5.....ïÄ.....
00000170 E0 00 00 00 07 00 00 00 07 00 00 00 A4 01 00 00 à.....ª....
00000180 35 00 00 00 00 02 00 00 F1 C4 02 00 00 00 03 00 5.....ñÄ.....
00000190 E7 00 00 00 52 00 00 00 30 00 00 00 10 02 00 00 ç...R...0.....
000001A0 35 00 00 00 00 02 00 00 E4 C4 02 00 00 00 03 00 5.....äÄ.....
000001B0 39 01 00 00 01 00 00 00 01 00 00 00 7C 02 00 00 9.....|....
000001C0 33 00 00 00 00 02 00 00 AD 66 00 00 00 00 01 00 3.....f.....
000001D0 3A 01 00 00 85 00 00 00 6E 00 00 00 E4 02 00 00 :.....n...ä...
000001E0 32 00 00 00 00 02 00 00 25 62 04 00 00 00 04 00 2.....šb.....
000001F0 BF 01 00 00 72 00 00 00 60 00 00 00 4A 03 00 00 ž...r...`...J...
00000200 50 00 00 00 00 02 00 00 0A 15 03 00 00 00 02 00 P.....
00000210 31 02 00 00 01 00 00 00 01 00 00 00 EC 03 00 00 1.....ì....
00000220 39 00 00 00 00 02 00 00 1D 26 04 00 00 00 07 00 9.....&.....
00000230 32 02 00 00 3A 00 00 00 32 00 00 00 60 04 00 00 2.....:...2...`...
00000240 37 00 00 00 00 02 00 00 95 E6 02 00 00 00 02 00 7.....*æ.....
00000250 6C 02 00 00 33 00 00 00 23 00 00 00 D0 04 00 00 1...3...#...Đ...
00000260 33 00 00 00 02 00 00 00 12 21 04 00 00 00 05 00 3

```

Illustration 1: Prefetch File Hex Dump

2.2 Parameters

A registry key exists to parameterize The Prefetcher. The setting parameters are stored in the Windows Registry at HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters. The EnablePrefetcher value can be set as follows: [3]

- 0 = Disabled
- 1 = Application prefetching enabled
- 2 = Boot prefetching enabled (default on Windows Server 2003 only)
- 3 = Application and Boot prefetching enabled (default)

2.3 Prefetch files

Prefetch files are stored in the directory: %SystemRoot%\Prefetch. In the same folder are also stored the following prefetch file types:

- *.pf, Prefetch files,
- Ag *.db and Ag *.db.trx, SuperFetch files,[5]
- Layout.ini,
- PfPre_* .db,
- PfSvPerfStats.bin

A Prefetch file contains the name of the application that it represents, a slash separator (-), an eight character hash value of the absolute path of the applications executable and a “.pf” extension. The file names must be represented by higher case letters, except the file extension. For example, a prefetch file name for the Calc program should be: CALC.EXE-4F89AB0C.pf. If the application is executed by two different paths, (e.g. C:\calc.exe and C:\Windows\System32\calc.exe), two different prefetch files will be generated in the prefetch folder. If an NTFS alternate data stream (ADS) is executed then the resulting file will also generate a prefetch file entry. According to MSDN, up to 128 prefetch files can be stored in the prefetch folder.

A PF consist of the following distinguishable elements:

- file header
- file metrics array
- trace chains array
- filename strings
- trailing data
 - volume information
 - file references
 - directory names
 - trailing data

2.4 Windows 10 prefetch files

Windows 10 prefetch files, have a different file format than the previous windows versions. No internal strings or text is available.[6] Since Windows 10, the information are stored in a compressed form in a MAM file similar to SuperFetch.[7]

2.4.1 MAM file format

A compressed Prefetch file consist of the following distinct elements:[7]

- File Header,
- Compressed Blocks,
- Block Terminator, (2 x 0-byte values)

The compression algorithm is Microsoft XPRESS Huffman (LZXPRESS). This compression algorithm is different than Microsoft XPRESS (LZ77+DIRECT2).

2.4.2 File Header

The file header has a size of 8 bytes. Three bytes with the signature 0x4d4d41 (MAM), one byte that identifies the compression algorithm used (0x4 in our case) [8] and a potential presence of a checksum. The next 4 bytes are the uncompressed size of the original buffer. The remaining data is what must be decompressed with RtlDecompressBufferEx.

2.5 Compression Engines

Windows NT comes with multiple built-in compression methods which are provided by the `RtlCompressBuffer`[9] and `RtlDecompressBuffer` functions:

- `COMPRESSION_FORMAT_LZNT1`, LZNT1 compression (LZ77)
- `COMPRESSION_FORMAT_XPRESS`, LZXPRESS compression (LZ77 + DIRECT2).
- `COMPRESSION_FORMAT_XPRESS_HUFF`, LZXPRESS with Huffman compression.

2.5.1 LZXPRESS Huffman

The LZXPRESS Huffman compressed data consists of multiple chunks. Each chunk consists of:[10][11]

- a prefix code table
- Huffman encoded bit stream

LZXPRESS Huffman prefix code table contains 512 x 4-bit prefix codes where the 4 LSB (Least Significant Bit) of byte 0 contain the prefix code for symbol 0, the 4 MSB (Most Significant Bit) the prefix code for symbol 1, etc. Where prefix codes:

- 0 - 255 represent their corresponding byte values;
- 256 - 511 represent compression tuples (size, offset).

2.5.2 Compression API

The **`RtlCompressBuffer`** function compresses a buffer and can be used by a file system driver to facilitate the implementation of file compression.[12] A bitmask is given as input that specifies the compression format and engine type. This parameter must be set to a valid bit-wise OR combination of one format type and one engine type.

<i>Compression Format</i>	<i>Signature</i>
<code>COMPRESSION_FORMAT_NONE</code>	0x0000
<code>COMPRESSION_FORMAT_DEFAULT</code>	0x0001
<code>COMPRESSION_FORMAT_LZNT1</code>	0x0002
<code>COMPRESSION_FORMAT_XPRESS</code>	0x0003
<code>COMPRESSION_FORMAT_XPRESS_HUFF</code>	0x0004

Table 1: Compression Algorithm Signature

<i>Compression Engine</i>	<i>Signature</i>
<code>COMPRESSION_ENGINE_STANDARD</code>	0x0000
<code>COMPRESSION_ENGINE_MAXIMUM</code>	0x0100
<code>COMPRESSION_ENGINE_HIBER</code>	0x0200

Table 2: Compression Algorithm Signature

The **`RtlCompressBuffer`** function takes as input an uncompressed buffer and produces its compressed equivalent provided that the compressed data fits within the specified destination buffer. To determine the correct buffer size for the **`WorkSpace`** parameter, the **`RtlGetCompressionWorkSpaceSize`** function is used. As a Windows API, the functions included can also be used by a third party application.

Starting from Windows XP, the prefetch folder, contains not only prefetch files, but also the layout.ini file. The layout.ini file is a list of the contents of the prefetch files, specifically the NTFS/MFT log sections that contain a list of files and their logical locations or paths. The entries in the layout.ini file are organized in the order in which they are loaded. The entries in the layout.ini file will then be moved or “reallocated” to a contiguous section of the hard drive, which will result in a faster recall time by the operating system. The process of moving the physical location of the files located in the layout.ini file occurs about every seventy-two hours when the Task Scheduler executes the defragmenter. The focus of the defragmenter is only on the contents of the layout.ini file and not the whole disk drive. Since these files are now physically located contiguously on the drive they will be read much faster.

3 Standby Memory

The Standby list, contains pages that have been removed from process working sets but are still linked to their respective working sets. The Standby list is essentially a cache. However, memory pages in the Standby list are prioritized in a range of 0-7, with 7 being the highest. A page related to a high-priority process will receive a high-priority level in the Standby list.[13]

For example, processes that are Shareable will be a high priority and pages associated with these Shareable processes will have the highest priority in the Standby list. If a process needs a page that is associated with the process and that page is now in the Standby list, the memory manager immediately returns the page to that process' working set. However, all pages on the Standby list are available for memory allocation requests from any process. When a process requests additional memory and there is not enough memory in the Free list, the memory manager checks the page's priority and will take a page with a low priority from the Standby list, initialize it, and allocate it to that process.

4 Compress Procedure

4.1 Decompress

For decompression of the Compressed prefetch files we used Windows-Prefetch-Parser. A Python script created to parse Windows Prefetch files: Supports XP - Windows 10 Prefetch files, created by Adam Witt a.k.a. PoorBillionaire.[14] It uses a modified version of Francesco Picasso's decompression script.[15] I pinpoint that you can't use on other OSES different from Windows, since it uses native API calls. Moreover you need Windows 8.1 at least, since the **RtlDecompressBufferEx** was introduced starting from that OS version.

4.2 Using the compression API

If an API exists that can decompress the prefetch file using the **Xpress Huffman**, another one should exist that can compress the contents of the prefetch file in order to for the prefetch files to exist. We used the **RtlCompressBuffer** function to re-compress compressed buffer we were provided by RtlDecompressBufferEx and then modified.[16]

<i>Error Codes</i>	<i>Signature</i>
STATUS_SUCCESS	0x00000000
STATUS_BUFFER_ALL_ZEROS	0x00000117
STATUS_INVALID_PARAMETER	0xC000000D
STATUS_UNSUPPORTED_COMPRESSION	0xC000025F
STATUS_NOT_SUPPORTED	0xC00000BB
STATUS_BUFFER_TOO_SMALL	0xC0000023

Table 3: Compression Algorithm Signature

We used "ctypes", which is a foreign function library for Python, to access the compression engine. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python. With ctypes we accessed *windll.ntdll.RtlCompressBuffer*, which is the compression library we used, and *windll.ntdll.RtlGetCompressionWorkSpaceSize*, to determine the correct size of the Workspace buffer. The *RtlCompressBuffer* and *RtlDecompressFragment* functions require an appropriately sized work space buffer to compress and decompress successfully. The *Workspace* parameter of the *RtlCompressBuffer* function must point to an adequately sized work space buffer. The *CompressBufferWorkSpaceSize* parameter of the *RtlGetCompressionWorkSpaceSize* provides this size.

We created the required buffers and passed them to the function *RtlCompressBuffer*. For the destination buffer, we provided a buffer with the size of the Uncompressed data. After the successful compression we can remove the empty bites at the end of the buffer by cutting the byte array with respect to the final compressed sized return by the function.

```

ntstatus = RtlCompressBuffer(
    USHORT(calgo),                # CompressionFormatAndEngine,
    ctypes.byref(ntUnCompressed), # Uncompressed Buffer
    ULONG(uncompressed_size),     # Uncompressed Buffer Size
    ctypes.byref(ntCompressed),   # Compressed Buffer
    ULONG(uncompressed_size),     # Compressed Buffer Size
    ULONG(chunk_size),           # Uncompressed Chunk Size
    ctypes.byref(ntFinalCompressedSize), # Final Compressed Size
    ctypes.byref(ntWorkspace)    # Work Space Size
)

```

After some trial and error we received a successful compression status code. We compared the initial compressed buffer with the compressed buffer we generated as a first validation. In order for the buffer to successfully be consumed by the system, the prefetch file header should also be added.

4.3 Generating the Header

The uncompressed prefetch file header, is a structure of 84 bytes and consists of the format version (with in our case, for windows 10 is the value of 30), the “SCCA” signature, the file size, the executable file name (as UTF-16 little-endian string with end-of-string character) and the Prefetch hash, with value should correspond with the hash in the prefetch filename. A couple of undistinguished values and flags are required, adding up to the size of 84 bytes.

The prefetch hash can be calculated with the SCCA 2008 hash function. [3] In order to hash the executable filename, the full path of the executable has to be determined and then converted into an upper-case Windows device path, e.g. ‘\DEVICE\HARDDISKVOLUME{volume id}\split-ed drive path’. Before the hash function is applied, the string must be converted into a UTF-16 little endian stream without a BOM (byte-order-mark) nor an end-of-string character. In short, comand we end up was:

```
file_for_hash = f'\DEVICE\HARDDISKVOLUME{volume_id}\path}'.upper().encode('utf-16-le').decode()
```

4.4 Manipulating the prefetch file

Till this point we were able to re-compress an uncompressed windows 10 prefetch file and reconstruct and attach the file header. With the prefetch data as plain text, we can edit its internals as we want. We can grub a random DLL file, rename it and then make sure it’s path and filename is included in the prefetch file.

Filename	Created Time	Modified Time	File Size	Process EXE	Process Path	Run Counter	Last Run Time	Missing Pr...
7ZG.EXE-0F8C4081.pf	20/12/2018 15:40	15/1/2019 12:52:38	23,018	7ZG.EXE	C:\PROGRAM FILES\7-Zip\7zG.exe	13	15/1/2019 12:52:28, 15/1/2019 12:45:36, 15/1/2019 12:52:38	No
7ZFM.EXE-6988961D.pf	15/1/2019 12:47:38	15/1/2019 15:21:06	16,894	7ZFM.EXE	C:\PROGRAM FILES\7-Zip\7zFM.exe	4	15/1/2019 12:53:34, 15/1/2019 12:53:14, 15/1/2019 12:53:34	No
LZXPRESS.EXE-6A33A8C...	20/12/2018 15:40	15/1/2019 15:35:47	13,187	DLI LPACT EVE	C:\Windows\System32\Dll...	6	15/1/2019 15:35:40, 15/1/2019 15:35:39, 15/1/2019 15:35:40	No

Illustration 2: WinPrefetchView view of 7ZIP prefetch

First we used WinPrefetchView by NirSoft, to verify if our new prefetch file can be parsed by another application. Our target executable was 7ZFM.exe. We modified one of it’s entries so it will reflect to a specific path/file. The reconstructed prefetch file was compressed with LZXPRESS Huffman algorithm and the header was added. WinPrefetchView was able to parse the file and recognize the entry we modified. The next step was to verify that Windows 10 system could successfully execute/load the prefetch file. We rebooted the machine and executed RamMap. Our newly modified DLL could be found Mapped in memory, listed as Standby.

shell66.dll	3/8/2018 11:39	Application extens	20.889 KB
ShellCommonCommonProxyStub.dll	12/4/2018 02:34	Application extens	395 KB
shellstyle.dll	12/4/2018 02:34	Application extens	1.129 KB
shfolder.dll	12/4/2018 02:34	Application extens	11 KB
shgina.dll	12/4/2018 02:34	Application extens	28 KB
ShiftUIS.uce	12/4/2018 02:34	UCE File	17 KB
shimgeng.dll	12/4/2018 02:34	Application extens	8 KB

Illustration 3: New DLL in folder (shell66.dll)

Filename	Full Path	Device Path
LOCALENLS	C:\Windows\System32\localenls	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
MPR.DLL	C:\Windows\System32\mpr.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
MSCTF.DLL	C:\Windows\System32\msctf.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
MSVCP110_WIN.DLL	C:\Windows\System32\MSVCP110_WIN.DLL	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
MSVCP_WIN.DLL	C:\Windows\System32\MSVCP_WIN.DLL	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
MSVCRT.DLL	C:\Windows\System32\msvcrt.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
NTDLL.DLL	C:\Windows\System32\ntdll.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
NTMARFA.DLL	C:\Windows\System32\ntmarfa.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
OLE32.DLL	C:\Windows\System32\ole32.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
OLEAUT32.DLL	C:\Windows\System32\oleaut32.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
POLICYMANAGER.DLL	C:\Windows\System32\POLICYMANAGER.DLL	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
POWERPROF.DLL	C:\Windows\System32\powerprof.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
PROFAPI.DLL	C:\Windows\System32\profapi.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
PROPSYS.DLL	C:\Windows\System32\propsys.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
RPCRT4.DLL	C:\Windows\System32\rpcrt4.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
RPCSS.DLL	C:\Windows\System32\rpcss.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
SECHOST.DLL	C:\Windows\System32\sechost.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
SHCORE.DLL	C:\Windows\System32\SHCore.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
shell66.dll	C:\Windows\System32\shell66.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
SHIMWAP.DLL	C:\Windows\System32\shimgapi.dll	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
TEXTINPUTFRAMEWORK.DLL	C:\Windows\System32\TEXTINPUTFRAMEWORK.DLL	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC
UIIMMPLIB.DLL	C:\Windows\System32\UIIMMPLIB.DLL	\\VOLUME{01643a7d47c65e36-a848b071}:\WINDC

Illustration 4: WinPrefetchView of 7ZIP prefetch detailed view, we can see that our new entry is included in the prefetch file.

RamMap - www.sysinternals.com

File Empty Help

Use Counts Processes Priority Summary Physical Pages Physical Ranges File Summary File Details

Path	Size	Physical Address	List	Type	Priority	Image	Offset
C:\windows\system32\catroot2\{127d0a1d-4ef2-11d1-8608-00c04fc295ee}\catdb	160 K						
C:\windows\assembly\nativeimages_v4.0.30319_64\mscorlib\60c56b4f785f2f9bead3537766610fa\mscorlib.ni.dll	4,904 K						
C:\users\v.vouvoitis\documents\utils\sysinternalsuite\autoruns.exe	128 K						
C:\program files\7-zip\7zfm.exe	440 K						
C:\windows\winsxs\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.17134.286_none_fb43982d30	1,452 K						
C:\users\v.vouvoitis\documents\utils\sysinternalsuite\rammap64.exe	616 K						
C:\users\v.vouvoitis\appdata\local\microsoft\windows\webcache\webcachev01.dat	4,548 K						
C:\windows\system32\windows.storage.onecore.dll	160 K						
C:\users\v.vouvoitis\appdata\local\spotify\browser\cache\data_3	128 K						
C:\windows\system32\shell66.dll	992 K						

Illustration 5: RamMap (Sysinternals) view of the shell66.dll. We can see that our file has successfully loaded into memory.

992 K						
0x5125000	Standby	Mapped File	1	Yes	0x1431600	
0x5F34000	Standby	Mapped File	4	Yes	0x43400	
0x6AC1000	Standby	Mapped File	4	Yes	0x54E600	
0x74FB000	Standby	Mapped File	4	Yes	0xE2400	
0xA522000	Standby	Mapped File	4	Yes	0x8C400	
0x140837000	Standby	Mapped File	4	Yes	0x40400	
0x1408DE000	Standby	Mapped File	4	Yes	0x2FC400	
0x1408F3000	Standby	Mapped File	4	Yes	0xFB400	
0x14199F000	Standby	Mapped File	4	Yes	0x5DC600	
0x141B3B000	Standby	Mapped File	4	Yes	0x35400	
0x141E17000	Standby	Mapped File	4	Yes	0x98400	
0x1422AD000	Standby	Mapped File	1	Yes	0x53F600	
0x14232D000	Standby	Mapped File	4	Yes	0x58400	
0x1423AC000	Standby	Mapped File	4	Yes	0x584600	
0x14273A000	Standby	Mapped File	4	Yes	0x36400	
0x14281E000	Standby	Mapped File	4	Yes	0x90400	
0x14299A000	Standby	Mapped File	0	Yes	0x504600	
0x142D04000	Standby	Mapped File	4	Yes	0xD8400	
0x142DAC000	Standby	Mapped File	1	Yes	0x53E600	
0x143618000	Standby	Mapped File	4	Yes	0x97400	
0x14361A000	Standby	Mapped File	1	Yes	0x1426600	
0x1436D2000	Standby	Mapped File	4	Yes	0x505600	
0x1436EB000	Standby	Mapped File	4	Yes	0x144400	
0x143793000	Standby	Mapped File	4	Yes	0x5E9600	
0x143794000	Standby	Mapped File	4	Yes	0x5E8600	
0x143795000	Standby	Mapped File	4	Yes	0x5E7600	
0x143796000	Standby	Mapped File	4	Yes	0x5E6600	
0x143797000	Standby	Mapped File	4	Yes	0x501600	
0x1437E1000	Standby	Mapped File	4	Yes	0x2B5400	
0x1437E2000	Standby	Mapped File	4	Yes	0x2B4400	
0x1437F7000	Standby	Mapped File	4	Yes	0xEA400	
0x143F35000	Standby	Mapped File	4	Yes	0x42400	
0x144218000	Standby	Mapped File	1	Yes	0x1424600	
0x144220000	Standby	Mapped File	4	Yes	0x8E400	
0x1442B8000	Standby	Mapped File	4	Yes	0x559600	
0x1442CD000	Standby	Mapped File	4	Yes	0x517600	
0x14481D000	Standby	Mapped File	4	Yes	0x91400	
0x145243000	Standby	Mapped File	4	Yes	0x2D400	
0x14529C000	Standby	Mapped File	1	Yes	0x536600	
0x14529E000	Standby	Mapped File	4	Yes	0x5DE600	
0x145316000	Standby	Mapped File	4	Yes	0x9B400	
0x145332000	Standby	Mapped File	4	Yes	0x50400	
0x1453D1000	Standby	Mapped File	4	Yes	0x508600	
0x14584E000	Standby	Mapped File	4	Yes	0x400	
0x14589C000	Standby	Mapped File	4	Yes	0x5E0600	
0x145929000	Standby	Mapped File	1	Yes	0x1435600	
0x1471BE000	Standby	Mapped File	4	Yes	0x548600	
0x147B9A000	Standby	Mapped File	4	Yes	0x5E2600	
0x147E0E000	Standby	Mapped File	4	Yes	0xA8400	
0x14823D000	Standby	Mapped File	4	Yes	0x33400	
0x148A28000	Standby	Mapped File	4	Yes	0x80400	
0x148A48000	Standby	Mapped File	4	Yes	0x24400	
0x148BD7000	Standby	Mapped File	4	Yes	0x4F4600	
0x148BEC000	Standby	Mapped File	4	Yes	0x143400	
0x1490A5000	Standby	Mapped File	1	Yes	0x53B600	

Illustration 6: RamMaop (Sysinternals), shell66.dll detailed view, loaded into standby memory

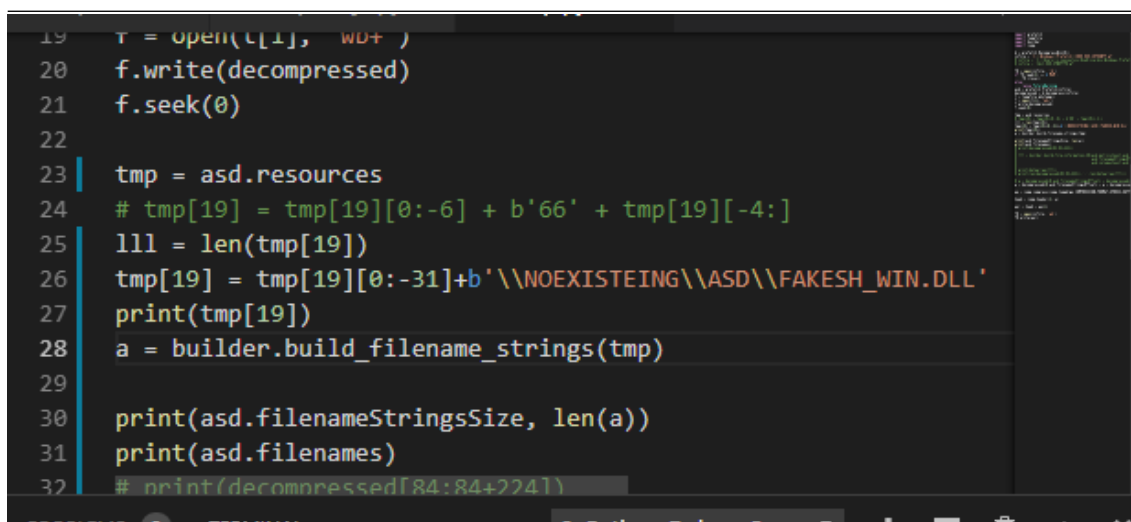
5 Scenarios

Eliminating the need for the use of external libraries is a nice to have for an attacker. Prefetch files are an artifact of an executable file engaging with its Eco-system and not a direct artifact of the executable. As such, if the executable is deleted, the Prefetch file persists.

Prefetch as far as we have understood so far has no control mechanisms, so the only way to control a fake entry is to MACE timestamps of NTFS which from what we noticed can be modified with Timestomp[17]. Timestomp's goal is to allow for the deletion or modification of timestamp-related information on files, such as MACE values. By successfully modifying timestamp-related information on prefetch files, the attacker can greatly decrease the possibility to be discovered.

5.1 Fake Entries

Most analysts are familiar with Prefetch files and how they can be useful to an examination. Prefetch files have a good bit of embedded metadata, and can be very useful during analysis. For example, you may look at the listing of files and something unusual may immediately jump out at you. If you include Prefetch file metadata in a timeline of a system, you should see a file access time for the executable "near" to when a Prefetch file for that executable is created or updated. If that's not the case, you may have an issue of time manipulation, or the original executable may have been deleted.



```
19 f = open(L[1], 'w+')
20 f.write(decompressed)
21 f.seek(0)
22
23 tmp = asd.resources
24 # tmp[19] = tmp[19][0:-6] + b'66' + tmp[19][-4:]
25 lll = len(tmp[19])
26 tmp[19] = tmp[19][0:-31]+b'\\NOEXISTEING\\ASD\\FAKESH_WIN.DLL'
27 print(tmp[19])
28 a = builder.build_filename_strings(tmp)
29
30 print(asd.filenameStringsSize, len(a))
31 print(asd.fileNames)
32 # print(decompressed[84:84+224])
```

Illustration 7: Adding a fake entry

Prefetch files also contain a number of strings, which are file names and full paths that point to modules used or accessed by the executable, and even some other files accessed by the executable. An attacker now is more than able to manipulate all this entries for his advantage.

Lets see for example a malicious application. The attacker can modify its execution times in order to hide the frequencies the application is used and delete those entries that pinpoint to unnatural file locations. For example, a malicious application can monitor its prefetch files and remove entries from the file list that are associated with its malicious activities. Additional we can replace file paths to pin point to a different file in a different location or even an non-existing file. When the correct file is required by the application, the system will first try to load it from the memory (as a prefetched entry) and upon failure it will loaded as normal from the disk drive, so missing or fake entries won't disturb the normal execution of the application.

Filename	Full Path
CODE.EXE-4EF96BFF.pf	C:\Users\Name\AppData\Roaming\Code\
CODE.EXE-4EF96C00.pf	C:\Users\Name\AppData\Roaming\Code\
CODEHELPER.EXE-EDE6DD44.pf	C:\Users\Name\AppData\Roaming\Code\
CODESETUP-STABLE-51B0B28134D5-36D19DFE.pf	C:\Users\Name\AppData\Roaming\Code\
CODESETUP-STABLE-51B0B28134D5-60E96BD4.pf	C:\Users\Name\AppData\Roaming\Code\
CODESETUP-STABLE-A622C65B2C71-AF037698.pf	C:\Users\Name\AppData\Roaming\Code\
CODESETUP-STABLE-A622C65B2C71-BBFA15DA.pf	C:\Users\Name\AppData\Roaming\Code\
CODESETUP-STABLE-C6E592B2B577-1673BEDC.pf	C:\Users\Name\AppData\Roaming\Code\
COMBATTE...BUNNER.EXE-03EAD00...pf	C:\Users\Name\AppData\Local\Programs\
EXTEND-NODE-611BC3497B48C490CB689EE12C5C5F25.CODE	C:\Users\Name\AppData\Roaming\Code\
EXTEND-NODE-BCD1763EDEF2A6492D52B6F2BB872851.CODE	C:\Users\Name\AppData\Roaming\Code\
FAKESH_WIN.DLL	C:\NOEXISTEING\ASD\FAKESH_WIN.DLL
FFMPEG.DLL	C:\Users\Name\AppData\Local\Programs\

Illustration 8: Fake Prefetch Entry to foreign file location

It should be noted that an application will, in most times, have dependencies located in the applications file path or in the system's path. So by linking file entries from the prefetch file to unrelated locations it will most probably alert the forensic investigator.

5.2 Storing Information

The prefetch file specification is based on earlier work on the format and was complimented by reverse engineering. No official documentation currently exist for this type of files. In this files, many unknown buffers exist than mostly contain empty values. An attacker can use these small buffers, usually with a fragmented size of less than a kilobyte, to store minimal information such as flags or small encryption keys. More over, the attacker can spread this information to multiple prefetch files in order to increase the amount of information he can store.

Let's imagine a situation where the attacker want's to hide a cryptographic key somewhere inside the windows Operating System. Prefetch files have a number of Buffers that store Unknown to this day values or even Empty values. These buffers are ranging from 8 bytes to 88 bytes. The maximum amount of bytes that can be stored it is possible to be able to be increased due to the structure of the prefetch file. All the prefetch tables and data entries have an offset relationship with other entries. So an attacker can probably add additional storage space between the prefetch entries just by increasing the prefetch entries offset accordingly from the corresponding tables.

From a forensics examiner's perspective, due to the undocumented nature of the prefetch file, it will require a great amount of effort just to even notice the information stored. No prefetch file examine utility will provide output with respect to these Unknown buffers, and even if a utility can provide the output we are not yet in position to recognize the maliciously stored information us we can't known what all of these buffers should normally store.

5.3 Loading into Memory

As we already know, files addressed in a prefetch file will be loaded in the standby memory for later use. Although it haven't been tested, an attacker can use the prefetch files to load malicious libraries into memory so he can use them later. Ideally, a program can parse the memory, like RamMap, to detect a malicious library loaded from the prefetch to the standby memory and execute them.

Let's review a scenario where after the initial compromise, the malicious application can download a malicious PE file (*The Portable Executable (PE) format is a file format for executables, object code, DLLs, FON Font files, and others used in 32-bit and 64-bit versions of Windows operating systems.*) that will be the malware. The downloader will check the registry if the prefetcher key is enabled and only then the prefetch file can be modified. We must also make an entry the layout.ini since it contains a list of all prefetched files. We can also examine

witch of the prefetch files belong to an application in the startup list or to a service to ensure that the malicious file will be loading into memory persistently.

We now have a malicious PE file loaded into standby memory. When an a forensic investigator tries to analyze the system, he will be able to trace that the malicious application is indeed loaded into the standby memory by the prefetcher. He will also be able to follow its path and examine the application itself. But it would be extremely unlikely to find the 3rd party malicious application that loads the PE file from memory.

6 Results

Since the attackers now have become intelligent, they even remove all these prefetch files from the system before leaving the system to remove any trail. Although prefetch files were mainly used for forensics investigations, there is clearly a possibility that an attacker can easily hide the activity of a malicious application by removing related entries from the prefetch files by decompressing them, modifying them and re-compressing them, all with the help of Windows compression and decompression APIs.

The content of each prefetch file provides rich information about the applications that were executed. There are two main sections of the prefetch file. The top, or first section, of the prefetch file contains the metadata of the file. The metadata includes the file name, file location, associated timestamps (file created, last accessed, and file modified), and the number of times the file was executed. The attacker is able to obscure his activities by modifying the aforementioned data. The modification can take place in any part of the file and the information that contains. Execution counts and execution times in addition to the volume information are critical for a forensic investigation but now an investigator has to double check their integrity by taking into account creation and modification time. Moreover, prefetch files can be used to load malicious code into memory.

Prefetch files do not include a mechanism that can prevent modification as their structural integrity is not critical for the operating system's operability. They are only designed to help boost operating systems and applications read times, but as it seems, not only they can not be trusted for an incident investigation but they can also be used to increase the attack surface of the operating system.

In addition to Prefetcher, Superfetcher, follows a similar architecture for the Superfetch (Ag*.db) file storage. Both files have the same MAM signature and are compressed with the same algorithm. Due to the fact that Superfetch files were introduced later in the Windows operating system architecture design, haven't yet been decoded sufficiently. It is possible that they can also be manipulated the same way we did with Prefetch files.

Prefetch files are a good source of evidence to determine the existence and execution of suspicious executables on a system. However, it is just one of the many Windows forensic artifacts that can help investigators understand what a user was doing on a system at a specific point in time. As a best practice, all Windows forensic artifacts should be examined and pieced together to see the bigger picture of an incident because as it turned out, one can no longer trust their integrity as they can be modified such as prefetch files generated from older Windows versions.

7 Python Source Code[18]

7.1 Python Imports

```
import sys
import ctypes
import enum
import binascii
import struct
```

7.2 Enumeration Classes

```
class CompAlgo(enum.Enum):
    COMPRESSION_FORMAT_NONE = 0x0000
    COMPRESSION_FORMAT_DEFAULT = 0x0001
    COMPRESSION_FORMAT_LZNT1 = 0x0002
    COMPRESSION_FORMAT_XPRESS = 0x0003
    COMPRESSION_FORMAT_XPRESS_HUFF = 0x0004
```

```
class CompEngi(enum.Enum):
    COMPRESSION_ENGINE_STANDARD = 0x0000
    COMPRESSION_ENGINE_MAXIMUM = 0x0100
    COMPRESSION_ENGINE_HIBER = 0x0200
```

```
class ErrorCodes(enum.Enum):
    STATUS_SUCCESS = 0x00000000
    STATUS_BUFFER_ALL_ZEROS = 0x00000117
    STATUS_INVALID_PARAMETER = 0xC000000D
    STATUS_UNSUPPORTED_COMPRESSION = 0xC000025F
    STATUS_NOT_SUPPORTED = 0xC00000BB
    STATUS_BUFFER_TOO_SMALL = 0xC0000023
```

7.3 Helper Function

```
def tohex(val, nbits):
    """Utility to convert (signed) integer to hex."""
    return hex((val + (1 << nbits)) % (1 << nbits))
```

7.4 Compression Method Function

```
def compress(
    algo: CompAlgo, uncompressed, engine: CompEngi =
    CompEngi.COMPRESSION_ENGINE_STANDARD, chunk_size: int = 4096
):
    calgo = algo.value | engine.value

    NULL = ctypes.POINTER(ctypes.c_uint)()
    SIZE_T = ctypes.c_uint
    DWORD = ctypes.c_uint32
    USHORT = ctypes.c_uint16
    UCHAR = ctypes.c_ubyte
```

```

ULONG = ctypes.c_uint32

# You must have at least Windows 8, or it should fail.
Try:
    RtlCompressBuffer = ctypes.windll.ntdll.RtlCompressBuffer
except AttributeError as e:
    sys.exit("[ - ] {e}\n"
            "[ - ] Windows 8+ required for this script to decompress Win10 Prefetch files")

RtlGetCompressionWorkSpaceSize = \
    ctypes.windll.ntdll.RtlGetCompressionWorkSpaceSize

ntCompressBufferWorkSpaceSize = ULONG()
ntCompressFragmentWorkSpaceSize = ULONG()

ntstatus = RtlGetCompressionWorkSpaceSize(USHORT(calgo),
ctypes.byref(ntCompressBufferWorkSpaceSize),
ctypes.byref(ntCompressFragmentWorkSpaceSize))

if ntstatus:
    raise EnvironmentError(f'Cannot get workspace size, err: '
        f'{tohex(ntstatus, 32)}:{ErrorCodes(tohex(ntstatus, 32)).name}')

uncompressed_size = len(uncompressed)

ntUnCompressed = (UCHAR * uncompressed_size).from_buffer_copy(uncompressed)
ntCompressed = (UCHAR * uncompressed_size)()
ntFinalCompressedSize = ULONG()
ntWorkspace = (UCHAR * ntCompressBufferWorkSpaceSize.value)()

ntstatus = RtlCompressBuffer(
    USHORT(calgo), # CompressionFormatAndEngine,
    ctypes.byref(ntUnCompressed), # Uncompressed Buffer
    ULONG(uncompressed_size), # Uncompressed Buffer Size
    ctypes.byref(ntCompressed), # Compressed Buffer
    ULONG(uncompressed_size), # Compressed Buffer Size
    ULONG(chunk_size), # Uncompressed Chunk Size
    ctypes.byref(ntFinalCompressedSize), # Final Compressed Size
    ctypes.byref(ntWorkspace) # Work Space Size
)
if ntstatus:
    raise ValueError(f'Cannot Compress Buffer, err: '
        f'{tohex(ntstatus, 32)}:{ErrorCodes(tohex(ntstatus, 32)).name}')
compressed = bytearray(ntCompressed)[:ntFinalCompressedSize.value]

return algo, compressed

```

7.5 Header Calculation Function

```

def header(algo, uncompressed):
    if algo is 4:
        sig = 0x44d414d
        header = struct.pack('<LL', sig, len(uncompressed))
        return header

```

7.6 SSCA 2008 Hash Function

```
def ssc_2008_hash_function(filename):
    hash_value = 314159
    filename_index = 0
    filename_length = len(filename)

    while filename_index + 8 < filename_length:
        character_value = ord(filename[filename_index + 1]) * 37
        character_value += ord(filename[filename_index + 2])
        character_value *= 37
        character_value += ord(filename[filename_index + 3])
        character_value *= 37
        character_value += ord(filename[filename_index + 4])
        character_value *= 37
        character_value += ord(filename[filename_index + 5])
        character_value *= 37
        character_value += ord(filename[filename_index + 6])
        character_value *= 37
        character_value += ord(filename[filename_index]) * 442596621
        character_value += ord(filename[filename_index + 7])

        hash_value = ((character_value - (hash_value * 803794207)) % 0x100000000)

        filename_index += 8

    while filename_index < filename_length:
        hash_value = (((37 * hash_value) + ord(filename[filename_index])) %
0x100000000)

        filename_index += 1

    return hash_value
```

8 Sample Prefetch File Structure Tables

8.1 File header

Offset	Size	Description
0	4	Signature 0x4d, 0x41, 0x4d, 0x04 (MAM\X04)
4	4	Total Uncompressed Data Size

8.2 Compressed Data Block

Offset	Size	Description
0	Var	LZXPRESS Huffman compressed data

8.3 Uncompressed Data Block

8.3.1 File Header

Offset	Size	Description
0	4	Format Version (<i>Win 10 is 30</i>)
4	4	Signature (SCCA)
8	4	<u>Unknown</u>
12	4	File Size
16	60	Executable Filename
76	4	Prefetch Hash
80	4	<u>Unknown</u>

8.3.2 File information

Offset	Size	Description
0	4	File Metrics Array Offset (0x00000130)
4	4	Number of file metrics entries
8	4	Trace chains array offset
12	4	Number of trace chains array entries
16	4	Filename strings offset
20	4	Filename strings size
24	4	Volumes information offset
28	4	Number of Volumes

32	4	Volumes information size
36	8	<u>Unknown</u>
44	64	Last run times
108	16	<u>Unknown</u>
124	4	Run count
128	4	<u>Unknown</u> (Seen: 1, 2, 7)
132	4	<u>Unknown</u> (Seen: 0, 3)
136	88	<u>Unknown</u>

8.3.3 File metrics array entry

Offset	Size	Description
0	4	<u>Unknown</u>
4	4	<u>Unknown</u>
8	4	<u>Unknown</u>
12	4	Filename string offset
16	4	Filename string number of characters
20	4	<u>Unknown</u>
24	4	NTFS file reference

Table 4: Contains metrics about the files loaded by the executable

8.4 Trace chains array

Offset	Size	Description
0	4	Next array entry index
4	4	Total block load count
8	1	<u>Unknown</u>
9	1	<u>Unknown</u>
10	2	<u>Unknown</u>

Table 5: A trace chain is similar to a File Allocation Table (FAT) chain where the array entries form chains and -1 (0xffffffff) is used to mark the end-of-chain. The chains in the trace chains array correspond with the entries in the file metrics array, meaning the first trace chain relates to the first file metrics array entry.

Bibliography

- 1: Russinovich, Mark; David Solomon, Microsoft Windows Internals (4th ed.), 2005
- 2: Zwiegincew; Arthur (Kirkland, WA), Walsh; James E. (Kirkland, WA), Pre-fetching of pages prior to a hard page fault sequence , October 14, 2003
- 3: Joachim Metz, Prefetch, 2018, <http://www.forensicswiki.org/wiki/Prefetch>
- 4: Mark Wade, Decoding Prefetch Files for Forensic Purposes, 12/08/2010, <https://www.forensicmag.com/article/2010/12/decoding-prefetch-files-forensic-purposes-part-1>
- 5: Joachim Metz, Windows SuperFetch database format, 2016,
- 6: Francesco Picasso, A first look at Windows 10 prefetch files, 2015,
- 7: Joachim Metz, Windows Prefetch File (PF) format, 2019,
- 8: Microsoft, winsdk-10, 2017,
- 9: MSDN, RtlDecompressBuffer function, 2018,
- 10: Microsoft Docs, Procedures, 2019,
- 11: Joachim Metz, Windows NT compression methods, 2016,
- 12: MSDN, RtlCompressBuffer function, 2018,
- 13: Greg Shultz, Investigate memory usage with Windows 7 Resource Monitor, 2010,
- 14: Adam Witt, Windows-Prefetch-Parser, 2016
- 15: Francesco Picasso, w10pfdecomp.py, 2015
- 16: Microsoft, RtlCompressBuffer function, 04/16/2018
- 17: James C. Foster and Vincent Liu, Blackhat briefings 2005, 2005
- 18: Vasileios Vouvoutsis, Prefetch Ninja, 2019