Digital Forensic Analysis on Prefetch Files

Exploring the Forensic Potential of Prefetch Files in the Windows Platform

Narasimha Shashidhar*[‡], Dylan Novak*

*Department of Computer Science, Sam Houston State University, Huntsville, Texas, USA.
{karpoor, dbn001}@shsu.edu

[‡]Corresponding author; Address: Department of Computer Science, 1803 Avenue I, Suite 214, Sam Houston State University, Huntsville, Texas 77341. Tel: (936) 294 – 1591, e-mail: *karpoor@shsu.edu*

Abstract- Prefetch files, like any other file in a file system, can be viewed from a digital forensic perspective to further a forensic investigation. Using appropriate tools and techniques available to a digital forensic examiner, we explore and investigate the potential of prefetch files and analyse what they have to offer from a digital forensic analysis perspective in an effort to contribute towards the rapidly advancing field of digital forensics. Windows' prefetch files are used to decrease the startup times of applications and are formatted in a manner to instruct application processes to load data and necessary libraries into memory that it needs before it is actually demanded. In other words, prefetch files help avoid a hard fault, thereby minimizing startup times. These files reside in the prefetch folder under the Windows installation directory of a system. This folder contains prefetch files for user and system applications as well as a ReadyBoot folder, a layout.ini file, and several database files. In this paper, we investigate the mechanism behind the creation and manipulation of prefetch files on a Windows machine. Next, we delve deep into the assembly code generated by the disassembler IDA PRO for the Windows executable ntkrnlpa.exe to find the Windows kernel processes responsible for the creation of these prefetch files and parse these prefetch files to better understand their forensic value.

Keywords- Prefetching; disassembly; digital forensics; reverse engineering; forensic analysis.

1. Introduction

Software developers and operating system designers have always been interested in speeding up the startup times of operating systems and their application software. To this end, Microsoft came up with an idea of dedicating a process to trace the data and libraries an application program most commonly needs upon startup and store this information in a format such that the process can easily read and load these necessary items into memory before it is actually needed, i.e., before it faults [1]. From then on, every time one launches a new program in Windows, a prefetch file gets created to speed up the time of the program's next startup. Now that we have this convenience, it begs the question of security and the intrinsic forensic value of these files. A file that contains instructions for several low level processes sounds pretty interesting. However, could this simple idea of decreasing startup-times lead to vulnerability in one’s system? Could we alter a prefetch file in such a way that it will launch another application with possible malicious intent? To answer these questions, we first decided to explore the contents and carefully parse a prefetch file into its constituent components. Then, we analyze the processes responsible for the reading and writing of a prefetch file in the Windows platform.

1.1 The Prefetch File

A. What's inside?

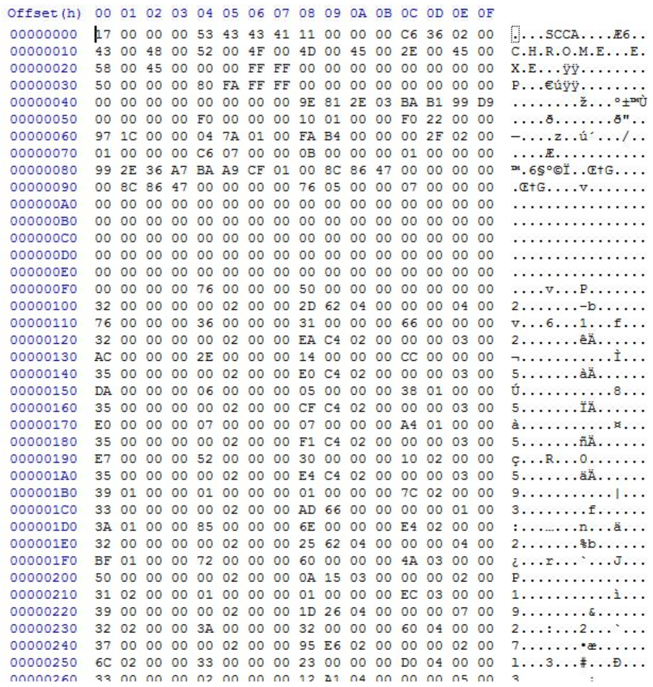
The entire purpose of a prefetch file's existence is to decrease the startup time of a program. That means that each individual file will hold data directly related to its respective program in a format similar to a list of instructions. Firstly, the name of a prefetch file is in the following format: *name of the application, an eight character hash of the location from where it was run, and a .pf extension*. The file itself will contain metadata such as the executable's name, the files and directories used by the application during the first 10 seconds of its execution, the size of the prefetch file, volume path, volume serial number, the run count, a timestamp of the creation time and the executable's last run time [2]. Apart from these staple items in a prefetch file, there is a rather large body of data containing instructions to load what the program most commonly uses at startup. This is the subject of our study in this work.

In Figure 1, we see the contents and the format of a typical prefetch file. This figure outlines the structure of CHROME.EXE-D999B1BA.pf, which is located in the directory *C:\Windows\Prefetch*. We used the popular *HxD* hex editor to view the raw format of the prefetch file in hopes to find out the contents and structure of the file so as to parse it further and enable forensic analysis.

Parts of this file are easier to interpret than others - such as the operating system (Win7 or Win XP), version number, and the size of the file. There are several researchers before us who have uncovered most of the staple metadata, but there is still a large body of the file that is yet to be parsed. This lead us to a decision of whether or not to go through the entire file and translate all values one-by-one, perhaps by trial and error, or whether we should disassemble and reverse engineer the very process responsible for creating this body of data and ultimately the file itself.

B. *What is prefetching?*

When a user selects an application to launch in the Microsoft Windows environment for the first time, a program called Windows Cache Manager traces data (mostly dynamic linked libraries, i.e., *dlls*) and other libraries that this application needs during start up. The data traced in memory is then saved in the prefetch file format and written into an appropriately named prefetch file by a kernel process called *NTKRNLPA.EXE*. Now, the next time the user selects that same application to launch, that prefetch file will be read by the same process and will load these necessary items into memory before it is actually demanded by the program. This act of prefetching is used for every application launched in Windows and even for the initial boot [1].



```
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....Æ6..
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...º±™Ù
00000050  00 00 00 00 F0 00 00 00 10 01 00 00 F0 22 00 00  ....ð.......ð"..
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  ....Æ...........
00000080  99 2E 36 A7 BA A9 CF 01 00 8C 86 47 00 00 00 00  ™.6§º©Ï..Œ†G....
00000090  00 8C 86 47 00 00 00 00 76 05 00 00 07 00 00 00  .Œ†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.......êÄ......
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...........¤...
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.......%b......
000001F0  BF 01 00 00 72 00 00 00 60 00 00 00 4A 03 00 00  ¿...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  l...3...#...Ð...
00000260  33 00 00 00 02 00 00 00 12 A1 04 00 00 00 05 00  3
```

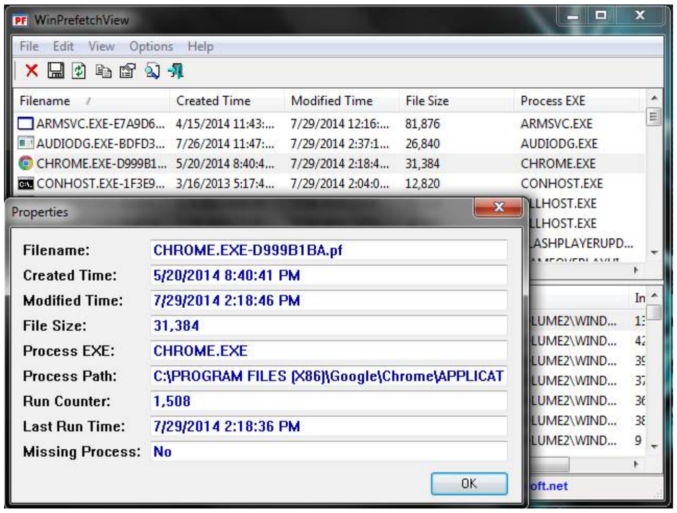
Fig 1. CHROME.EXE-D999B1BA.PF IN HXD

C. *Why is it useful?*

Prefetch files make it easier for software applications and programs to find what they need on the hard disk. Without them, every program would have to wait on the performance of the hard drive to find any piece of data it needs at time of startup or hard fault. In this day and age, programs use a lot more memory and libraries than before. So, prefetch files are useful when it comes to speeding up load times. In this work, we explore the question of how these files might be valuable in a forensic investigation. The answer clearly lies in the contents of the file. Forensic specialists have developed programs specifically for prefetch files like WinPrefetchView (Fig 2.) by NirSoft. These programs easily scan the prefetch file selected and then parse out all known metadata items in a simple format for users to read and interpret.

Since the Windows Cache Manager traces up to ten seconds of activity after an application has launched, we can use WinPrefetchView to see which files and directories the user had accessed after initial launch of the application [3]. This can be very useful in the field of digital forensics.

Prefetch files have been shown to provide significant metadata, or "data about data". Looking at the raw hexadecimal values in a hex editor gives us a detailed look at the entire file format while WinPrefetchView just gives us basic glimpse at the most significant and common properties that digital forensic examiners are interested in (and are potentially aware of).



Filename	Created Time	Modified Time	File Size	Process EXE
ARMSVC.EXE-E7A9D6...	4/15/2014 11:43:...	7/29/2014 12:16:...	81,876	ARMSVC.EXE
AUDIODG.EXE-BDFD3...	7/26/2014 11:47:...	7/29/2014 2:37:1...	26,840	AUDIODG.EXE
CHROME.EXE-D999B1...	5/20/2014 8:40:4...	7/29/2014 2:18:4...	31,384	CHROME.EXE
CONHOST.EXE-1F3E9...	3/16/2013 5:17:4...	7/29/2014 2:04:0...	12,820	CONHOST.EXE

Filename:	CHROME.EXE-D999B1BA.pf
Created Time:	5/20/2014 8:40:41 PM
Modified Time:	7/29/2014 2:18:46 PM
File Size:	31,384
Process EXE:	CHROME.EXE
Process Path:	C:\PROGRAM FILES (X86)\Google\Chrome\APPLICAT
Run Counter:	1,508
Last Run Time:	7/29/2014 2:18:36 PM
Missing Process:	No

Fig 2. WINPREFETCHVIEW

WinPrefetchView properties list:

- Filename – CHROME.EXE-(Hash value).pf
- Created time
- Modified time
- File size on disk
- Process EXE – CHROME.EXE
- Process Path - C:\PROGRAM FILES\...
- Run Counter – Number of times launched
- Last Run Time
- Missing Process

We contend that these are just basic properties we expect to find in every prefetch file that the forensic community is aware of at the moment. There is much more, however, that we are able to see in the raw file using a binary editor. This task of course requires translating a series of binary data via experimentation, trial-and-error, and using educated guesses. The good news is that there has been a lot of contribution towards the translation of the different offset addresses in the prefetch file.

b) Other notable properties not found in WinPrefetch View

- Windows version – XP, Vista, Win7, Win8
- Volume device path/length/creation time
- Directory name and number of characters

D. The layout file - Layout.ini

Let us now look at the file *Layout.ini* contained in the prefetch directory. Layout.ini is a file found in the Windows prefetch directory and is similar to the idea of a prefetch file but instead relates to the boot process of the machine. The task scheduler calls Windows disk defragmenter every three days and accumulates a list of common files referenced by the system during boot up. These files are listed in the Layout.ini file as instructions to preload certain files before they are needed during boot up, ultimately decreasing the time required to boot up the system. The disk defragmenter checks and reorders this list every three days while the machine is idle to update what the system commonly uses during the boot process [3].

1.2 Prior and Related Work

One of the earliest articles discussing prefetch files and their potential application in Digital Forensic Analysis is by Mark Wade [3]. In this article, he explores the many different forensic artefacts that can be discovered from Windows prefetch files. This article, while being comprehensive, lacks academic scrutiny and seems to be composed using commercial software documentation, blogs, and Wikipedia. As a result of the lack of scrutiny, Wade's article inadvertently introduces misconceptions. The lack of a comprehensive body of academic research on prefetch files also results in researchers and commercial software vendors taking for granted certain evidentiary and forensic value of such data. The article by Shiaeles et al. explores the effectiveness of on-scene triage tools used by forensic investigators [6].

Another work done by Lee et al. explores on-site investigation methodology for incident response [7] in Windows environments and includes reference to prefetch files in the same way that other articles do. An April, 2010 blog post by Yogesh Khatri provides much valuable insight as to the format of the application prefetch files; however, it does not fully describe the structure [8]. In contrast, the third edition of Harlan Carvey's book, *Windows Forensic Analysis Toolkit,* is careful to describe prefetch files and the artefacts therein as interesting indicators [9], without making definitive statements about their evidentiary value or drawing conclusions without proper foundation.

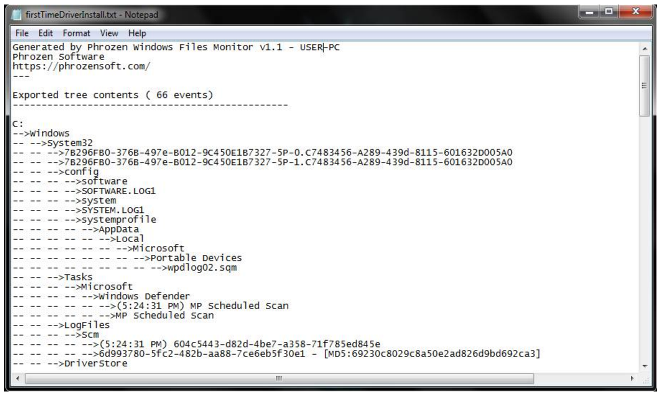
Lastly, McQuaid's August, 2014 blog post, in addition to reinforcing the concept that prefetch files can be an important resource for digital investigations in general, states that they can also be useful for timeline analysis [10]. A forensic investigator can potentially identify other files that were created or modified during the same timeframe as a malicious program execution, for example, and subsequently use this information to determine the root cause of an incident [10]. In conclusion, it appears from the existing body of work that a particular emphasis is being placed on what data artefacts exist in the various prefetch files at a particular point in time, and less emphasis on the processes behind their creation.

2. Our Goal

In this section, we motivate our decision to explore the potential of prefetch files for forensic analysis. Initially, our goal was to extract incriminating evidence left behind in RAM or on a hard drive from a session where a criminal might have used a USB mobile application through a portable USB drive. After a series of experiments and some research, we discovered that much of this evidence seemed to reside in the prefetch files. This led us to explore the potential of prefetch files as it relates to a forensic investigation. These files have the ability to control what is loaded into memory when a user starts up an application. Therefore, it is reasonable to assume that the remnants of portable device application activity can be found using prefetch file analysis.

A. *USB flash drive digital forensics*

In the beginning, we wanted to test, explore, and extract the digital forensic evidence a USB flash drive and the portable applications that it hosts will leave behind on a typical installation of a Windows operating system (XP and 7). The idea was to find any files, snapshots, or any kind of data left behind by the driver, user, or by the portable application itself.


firstTimeDriverInstall.txt - Notepad
File Edit Format View Help

```
Generated by Phrozen Windows Files Monitor v1.1 - USER-PC
Phrozen Software
https://phrozensoft.com/
---

Exported tree contents ( 66 events)
------------------------------------------------

C:
-->Windows
-- -->System32
-- -- -->7B296FB0-376B-497e-B012-9C450E1B7327-5P-0.C7483456-A289-439d-8115-601632D005A0
-- -- -->7B296FB0-376B-497e-B012-9C450E1B7327-5P-1.C7483456-A289-439d-8115-601632D005A0
-- -- -->config
-- -- -- -->software
-- -- -- -->SOFTWARE.LOG1
-- -- -- -->system
-- -- -- -->SYSTEM.LOG1
-- -- -- -->systemprofile
-- -- -- -- -->AppData
-- -- -- -- -- -->Local
-- -- -- -- -- -- -->Microsoft
-- -- -- -- -- -- -- -->Portable Devices
-- -- -- -- -- -- -- -- -->wpdlog02.sqm
-- -- -->Tasks
-- -- -- -->Microsoft
-- -- -- -- -->Windows Defender
-- -- -- -- -- -->(5:24:31 PM) MP Scheduled Scan
-- -- -- -- -- -->MP Scheduled Scan
-- -- -->LogFiles
-- -- -- -->Scm
-- -- -- -- -->(5:24:31 PM) 604c5443-d82d-4be7-a358-71f785ed845e
-- -- -- -- -->6d993780-5fc2-482b-aa88-7ce6eb5f30e1 - [MD5:69230c8029c8a50e2ad826d9bd692ca3]
-- -- -->DriverStore
```

Fig 3. PHROZEN WINDOWS FILES MONITOR

As an initial step we mapped out all the *events* that happen on the operating system when a USB storage device is connected to Windows. What we found was that Windows does a more than "wellenough" job of monitoring events and activities that happen in the background. One session of using Phrozen Windows Files Monitor, a popular tool used to monitor activity and changes on storage devices e.g. Hard Disk Drive and connecting a USB flash drive to the machine showed us what the Windows OS actually changes, reads, or writes during this process. We noticed a couple of entries made in the *System32* directory and activity in some default configuration and temporary files. Of the entire list of changes on the live capture that we noticed, the changes made to the folder *C:\Windows\Prefetch* stood out.

B. Prefetching Potential

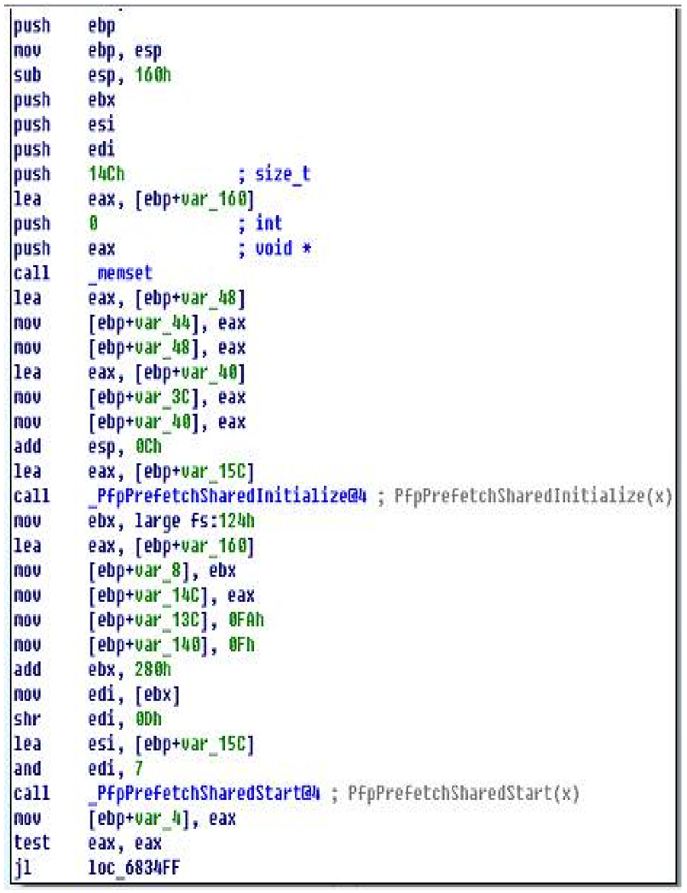
Looking further into the prefetch directory, we find a list of **.pf* files for every user application that was previously run on the system. We then opened *CHROME.EXE-D999B1BA.pf* in HxD Hex editor to look at the file format as discussed earlier. After analyzing the contents, we wondered if these prefetch files contained valuable forensic information. We wanted to find out what processes reads/writes these files, the manner in which they are read/written, and if we could alter them? Other questions that came to our mind, but we haven't explored in this current work, is whether prefetching can be used towards anti-forensics and any potential vulnerability that this might introduce on a host system. In theory, it is conceivable that a cybercriminal could recreate the functions used and called in the process responsible for creating prefetch files, and write instructions themselves to alter application settings, configurations, and what executes. This means another process/application could easily replace an instruction and have a malicious application loaded into RAM as well.

3. Reverse Engineering and Disassembly

Finding the answers to our above mentioned questions meant that we would have to delve deep into the process of how prefetch files are created, manipulated and saved on a Windows XP computing system. A core Windows component by the name of *ntkrnlpa.exe*, a kernel process, is responsible for reading, writing and manipulating prefetch files based on the instructions from the Windows Cache Manager. In order to learn how ntkrnlpa.exe works and discover exactly what it does, we had to disassemble the executable and analyze the assembly code that was generated. This is a natural thing to do in Windows forensics, since many processes and events in the Windows operating system are not documented by Microsoft in much detail.

A. Disassembling Ntkrnlpa.exe

First, we noticed that Ntkrnlpa.exe is the kernel executable responsible for writing prefetch files in Windows. This process takes data traced and monitored by the Windows Cache Manager and writes it into prefetch file format.



```
push    ebp
mov     ebp, esp
sub     esp, 160h
push    ebx
push    esi
push    edi
push    14Ch            ; size_t
lea     eax, [ebp+var_160]
push    0               ; int
push    eax             ; void *
call    _memset
lea     eax, [ebp+var_48]
mov     [ebp+var_44], eax
mov     [ebp+var_48], eax
lea     eax, [ebp+var_40]
mov     [ebp+var_3C], eax
mov     [ebp+var_40], eax
add     esp, 0Ch
lea     eax, [ebp+var_15C]
call    _PfpPrefetchSharedInitialize@4 ; PfpPrefetchSharedInitialize(x)
mov     ebx, large fs:124h
lea     eax, [ebp+var_160]
mov     [ebp+var_8], ebx
mov     [ebp+var_14C], eax
mov     [ebp+var_13C], 0FAh
mov     [ebp+var_140], 0Fh
add     ebx, 280h
mov     edi, [ebx]
shr     edi, 0Dh
lea     esi, [ebp+var_15C]
and     edi, 7
call    _PfpPrefetchSharedStart@4 ; PfpPrefetchSharedStart(x)
mov     [ebp+var_4], eax
test    eax, eax
jl      loc_6834FF
```

Fig 4. EXAMPLE OF IDA PRO'S DIS-ASSEMBLY

To disassemble ntkrnlpa.exe, we used the popular program from Hex-Rays called IDA (Interactive Disassembler) Pro Freeware. IDA Pro is able to disassemble Windows' executables and return the results to the user in assembly code as best as possible. After the disassembly is complete, we are given the entire list of functions and procedures used in ntkrnlpa.exe as well as any imports and exports. We then analyze the results and attempt to reverse engineer the kernel process one function at a time.

Our part in reverse engineering involves inspecting each relevant function in the hope of obtaining a better understanding of another related, relevant function. In the end, we aim to be left with a map of function calls representing the chain of processes that take place behind the creating, reading, and writing of prefetch files. We then take this information and translate it further into a higher level language which in turn mirrors what the original process does. This takes a complete understanding of the reverse engineering process and how ntkrnlpa.exe really operates on prefetch files.

B. Disassembly

Disassembling ntkrnlpa.exe left us with several thousands of lines of assembly code and a rather large list of functions. After some time of searching through these functions we were able to come up with a list of, what we believe are, the most significant functions which we hope will lead to further disassembly of the prefetch process.

It then took some time to narrow our results down to functions we thought were relevant to prefetch files and the process of prefetching. We accomplished this by finding the prefix "PF" in front of the function names (an educated guess that turned out to be right) and cross-referencing through their tangent, interstitial called functions. From there, we tried to map out the different function calls and through an iterative trial-and-error process, able to recreate certain functions. We demonstrate a couple of our attempts below.

List of functions related to .pf files in NTKRNLPA.EXE:

- [!]PfSnBeginAppLaunch
- PfPrefetchRequestVerify
- PfPrefetchRequestVerifyPath
- PfPrefetchRequestVerifyRanges
- [!]PfSnGetPrefetchInstructions
- PfSnIsHostingApplication
- PfpGetParameter
- PfpLogApplicationEvent
- PfpLogEventRequest
- PfpPrefetchRequest
- PfGenerateTrace
- PfFileInfoNotify
- PfLogFileDataAccess
- PfPowerActionNotify
- [!]PfCalculateProcessHash
- PfCheckDeprioritizeFile
- PfLogFileDataAccess
- SeLocateProcessImageName
- ExfAcquirePushLockShared
- KeInitializeEvent
- [!]PfSnScanCommandLine

4. The prefetching process: ntkrnlpa.exe

How are prefetch files created? To answer this, we must analyze the resulting disassembly code of ntkrnlpa.exe. The following steps explain how the reverse engineering process was performed and then we describe the insights we got into the prefetching process.

STEP 1: Starting Point

Since ntkrnlpa.exe is a huge executable, it uses hundreds of functions to perform operations that are not always related to prefetch files. Therefore, we must narrow our search of functions to those relating to prefetching. Initially scanning through the list, we began to notice a prefix pattern of “pf” to several functions. A quick string search of “pf” in sorted alphabetical order resulted in a list of functions we have assumed to be responsible for creating and manipulating prefetch files. From here, we attempted to find the best starting point to the initial creation of a prefetch file.

STEP 2: *PfProcessCreateNotification*

One of the functions that made sense to start working on was the subroutine PfProcessCreateNotification. We assumed this function has something to do with the initial creation of a prefetch file and naturally made for a good starting point. As the function PfProcessCreateNotification is called, it tests a flag possibly verifying if a prefetch file previously exists. If the result of the condition is not zero then the parameters and the value in the source index register gets pushed onto the stack and PfCalculateProcessHash is called. After the hash is produced, PfSnEnablePrefetcher is tested and followed by a condition ***jz***. The statement looks like;

```
[cmp   _PfSnEnablePrefetcher,  0]
[jz (address in memory)]
```

This appears often in a few of the other functions related to prefetch files. If the condition is not zero, a local variable gets moved into *edx* and the current value of the source index register gets moved into *ecx* before *PfSnBeginAppLaunch* is called.

STEP 3: *PfSnBeginAppLaunch*

This function specifically has quite a bit of code to it and still has some knowledge gaps that we are yet to completely understand. As it starts, multiple variables and fields are declared and initialized. The operations mainly consist of moving *esi* into the address of the stack pointer plus, 80h plus the value of the corresponding variable. Following the last *mov* statement is a condition similar to the one we saw in PfProcessCreateNotification. It again tests the result of the PfSnEnablePrefetcher and proceeds with a series of comparisons and jump conditions. We assume this must be related to the creation of the prefetch file header since the operations involve testing different strings with the source index register. The series of conditions funnel down into a single block again and then calls *PfSnIsHostingApplication* followed by a call to *PfSnScanCommandLine*.

If PfSnScanCommandLine returns a value, it is saved as a variable and is prepared to be passed as a parameter, along with two others, to the upcoming function which is appropriately named PfSnGetPrefetchInstructions.

STEP 4: *PfSnGetPrefetchInstructions*

The function *PfSnGetPrefetchInstructions* begins life by pushing all necessary items onto the stack and then starts a loop. The loop begins by moving the contents of eax into dx then incrementing eax twice. A compare statement follows, comparing the source index with dx. If the two are not equal we return back to the beginning of the loop and repeat until the condition is met. We believe this process records the necessary **.dll* files being called for the respective executable. The Table below (Table 1) succinctly outlines these steps.

Table 1. Prefetch Function Table

Prefetch Function	Estimated Description
PfProcessCreateNotification	Checks for existing prefetch file and sets a flag for other functions to reference.
PfCalculateProcessHash	Creates a hash value sensitive to the length of the file.
PfSnEnablePrefetcher	Used as a flag often to test if other functions may proceed.

PfSnBeginAppLaunch	May be responsible for the creation of the header.
PfSnScanCommandLine	Reads any strings from the command line and saves them as variables.
PfSnGetPrefetchInstructions	Records the names of the required .dll's by the executable.
PfPrefetchRequestVerifyPath	Validates the path of the recorded files and executables

a) PfCalculateProcessHash

This was the first function we began to take an in-depth look at. We believe that this functin is responsible for identifying the application's location via the path on the HDD. It then uses the algorithm outlined in Figure 5 to create the hash for the prefetch file. This script was originally written in Perl by Blaszczyk [4]. He too attempted to disassemble ntkrnlpa.exe and reverse engineer the PfCalculateProcessHash function.

Fig 5. PFCALCULATEPROCESSHASH IN PERL

```
sub hash_xp
{
  my $devpath_u = shift;
  my $hash = 0;
  for (my $i=0; $i<length($devpath_u);
$i++)
  {
      my $char =
ord(substr($devpath_u,$i,1));
      $hash = ( ($hash * 37) + $char ) %
4294967296;
      #print STDERR
sprintf("%08lX",$hash).'
'.substr($devpath_u,$i,1)."\n";
  }
  $hash = ($hash * 314159269) %
4294967296;

  $hash = 0x100000000-$hash if
($hash>0x80000000);
  $hash = (abs($hash) % 1000000007) %
4294967296;
  return $hash;
}
```

b) PfSNBeginAppLaunch

After analyzing PfCalculateProcessHash further we stumbled upon another process of interest, i.e, PfSNBeginAppLaunch. The name of the function encouraged us to believe that we were on the right path. Under this function, we found several other references and calls to related and interesting functions such as: *PfSNGetPrefetchInstrunctions*, *PfSnBeginTrace*, *PfSnScanComandLine*, and *PfSnPrefetchScenario*. These functions were the most notable in our hopes of bringing us closer to our goal of understanding the background processes responsible for prefetching in Windows.

C. Window's prefetch file format

Inside our HxD hex editor program (Figure 1.), we could see the raw format of a prefetch file (specifically CHROME.exe-d999b1ba.pf). From there, we were then able to cross reference any and all known properties, so as to not duplicate our efforts, from the prefetch format table available on the popular Digital Forensics Wiki Portal www.*forensicswiki.org*, relating to their position as an offset address and verifying the values in hexadecimal ourselves [5].

a) File Header

This table below represents a general prefetch file header. This header contains data that we ended up using to retrieve carved data.

FILE HEADER FORMAT

Offset	Length	Notes
0x0000	4	Format Version of the Windows OS 17 – Win XP/2003 23 – Win Vista/Win7 26 – Win8.1
0x0004	4	SCCA signature
0x0008	4	Unknown- Values observed 0x0F-Win XP 0x11 – Win 7/8
0x000C	4	Prefetch file size
0x0010	60	Corresponding name of the executable - up to 29 characters in length
0x004C	4	Corresponding prefetch hash value
0x0050	4	Unknown – This could be a flag property given most values are 0 and 1

b) File Information

Following the file header is a block of file information that is dependent on the version of windows. Three different versions exist to provide directions to the start, length, and end of the different sections. File information version 17 is 68 bytes in size, version 23 is 156 bytes in size, and version 26 is 224 bytes in size. Closer to the end of the file information block, lies the execution last run time and execution counter.

This is a representation of version 26 file information block from offsets 0x0054 to 0x00DC.

FILE INFORMATION FORMAT

Offset	Length	Notes
0x0054 -0x0074	36	Consists of the length, number of entries, and offset locations for sections A, B, C, D
0x0078	8	Unknown
0x0080	8	Latest execution time of executable
0x0088	56	Most recent execution time of executable
0x00C0	16	Unknown
0x00D0	4	Execution counter
0x00D4 - 0x00DC	96	Unknown

c) SECTION A – Metrics array

After the file information begins the first section, 'A', which contain the metrics entry records. Version 17 consists of 20 bytes in size and version 23 and 26 are both 32 bytes in size. The offsets vary in numerical value depending on the length or version of the fields before it. These metric array entries correspond with the entries in the file name strings [5].

SECTION A – METRICS ARRAY

Offset	Length	Notes
0	4	Start time in ms
4	4	Duration in ms
8	4	Average duration in ms
12	4	File name string offset
16	4	File name string number of characters
20	4	Unknown
24	8	NTFS file reference

d) SECTION B – Trace chains array

Section B consists of a 12 byte block of entry records same for all three versions. The offsets are also relative to the previous versions and size of blocks beforehand.

SECTION B – TRACE CHAINS ARRAY

Offset	Length	Notes
0	4	Next array entry index
4	4	Total block load count. Number of blocks fetched
8-10	4	Unknown

e) SECTION C -Filename strings

We have learned from previous work that Section C mainly consists of an array of little-endian formatted strings with end-of-string characters [5].

f) SECTION D – Volumes information

Section D contains information on the volumes related to the directories accessed during prefetching. Multiple subsections are held in section D proportional to the amount of volumes referenced. If only one volume is referenced then there will only be one subsection in section D [5]. Again both version 23 and 26 are identical consisting of 104 bytes while version 17 is 40 bytes in size.

Section D – Volumes Information

Offset	Length	Notes
+0	4	Offset to volume device path
+4	4	Length of volume device path
+8	8	Volume creation time
+10	4	Volume serial number of volume
+14	4	Offset to sub section E
+18	4	Length (size) of sub section E
+1C	4	Offset to sub section F
+20	4	Number of string in sub section F
+24	4	Unknown

The two sub sections referenced in section D are the NTFS file references (sub section E) and a block of Directory Strings (sub section F).

1) NTFS file reference

Offset	Length	Notes
0	6	MFT entry index
6	2	Sequence number

2) Directory Strings

Offset	Length	Notes
0	2	String number of characters of directory name.
2		Directory name as Unicode followed by end-of-string character.

4. Conclusions and Future Work

In this paper, we initiated a study that explored the potential of prefetch files as it relates to a forensic investigation. To the best of our knowledge, we are the first to have identified several interesting prefetching processes and have demonstrated a clear approach to parsing and understanding their operations. In the future, we intend to explore the prefetching process in greater detail and in particular complete parsing the missing and unknown sections of the prefetch file. We believe that most of the unknown and missing properties/information consists of flags and/or variations of properties from previous versions. We are also continuing with our work on reverse engineering the ntkrnlpa.exe process of reading/writing prefetch files. We hope to be able to identify the functions and the set of instructions used, and analyze any information we can to learn even more about prefetch files. After completion, we will attempt to take what we have learned about prefetching and apply this knowledge towards USB device digital forensics and how we might be able to extract digital evidence and artifacts left behind by portable applications running on a portable USB drive. We would like to note that there are several unanswered questions that still remain to be explored. For instance, it is conjectured that one might be able to inject entries into a prefetch file so as to force the operating system loader to load malicious or otherwise unnecessary libraries (*.*dll*s) into memory without being detected. Lastly, it is worth mentioning that to begin the disassembly process of ntkrnlpa.exe, one might begin searching for the "SCCA" signature of a prefetch file and identifying relevant functions.

Acknowledgements

The authors would like to thank the support of the EURECA Center at Sam Houston State University for funding this research.

References

[1] Shanno Cepeda, "What you Need to Know about Prefetching," *Intel® Developer Zone*. Latest Access Time for the website is February 8th 2015. https://software.intel.com/en-us/blogs/2009/08/24/what-you-need-to-know-about-prefetching.

[2] Forensics Wiki, "Prefetch," Latest Access Time for the website is Jan16th 2015. http://www.forensicswiki.org/wiki/Prefetch

[3] Mark Wade, "Decoding Prefetch Files for Forensic Purposes: Part 1." *DFI News*. Latest Access Time for the website is August 8th 2014. http://www.dfinews.com/articles/2010/12/decoding-prefetch-files-forensic-purposes-part-1.

[4] Adam Blaszczyk, "Hexacorn | Blog," *Hexacorn Ltd Blog Posts RSS*. Latest Access Time for the website is July 26th 2014. http://www.hexacorn.com/blog/2012/06/13/prefetch-hash-calculator-a-hash-lookup-table-xpvistaw7w2k3w2k8

[5] Joachim Metz, "Analysis of SCCA," *Windows Prefetch File (PF) format* 1 (2011): 25.

[6] Shiaeles Stavros, Anargyros Chryssanthou, and Vasilios Katos, "On-scene triage open source forensic tool chests: Are they effective?," *Digital Investigation* 10.2 (2013): 99-115.

[7] Keungi Lee, Changhoon Lee, and Sangjin Lee, "On-site investigation methodology for incident response in Windows environments," *Computers & Mathematics with Applications* 65.9 (2013): 1413-1420.

[8] Yogesh Khatri, "Windows Prefetch File", [Online], Available: http://www.swiftforensics.com/2010/04/the-windows-prefetchfile.html, April, 2010, [Latest Access Time for the website is Nov 14, 2014]

[9] Harlan Carvey, "Windows forensic analysis DVD toolkit", *Third edition. Syngress*, 2012, pp 88-92.

[10] Jamie McQuaid, "Forensic Examination of Prefetch Files in Windows." [Online], Available: http://www.magnetforensics.com/forensic-analysis-of-prefetch-files-in-windows/, Aug 6, 2014 [Latest Access Time for the website is Feb 4, 2015]