Firefox Cache Format and Extraction

POSTED BY JOHN RITCHIE · MARCH 9, 2012 · 4 COMMENTS

FILED UNDER BROWSER FORENSICS, DATA RECOVERY, FIREFOX, FORENSIC METHODOLOGY

Introduction

In the forensic lab where I work, we frequently investigate malware-infected workstations. As our user population started shifting from Internet Explorer to Firefox, we observed that one of our favorite forensic tools, Kristinn Gudjonsson's log2timeline, wasn't able to provide as much data for Firefox as it was for IE. The missing component was cache data; log2timeline was capable of parsing IE cache but not Firefox. In order to fix this deficit and contribute to log2timeline, I decided to write a log2timeline module for the Firefox cache. During the course of writing that module (ff_cache.pm – available in log2timeline 0.62), I researched how the Firefox cache works, wrote a tool to extract data from it (ff_cache_find), and learned traits of Firefox that have implications for forensic acquisition and analysis.

This paper describes the format and functionality of the Firefox cache in order to help forensic investigators understand how to make the most of Firefox evidence. Although there are several available tools, both free and commercial, to analyze the Firefox browser cache, there is little material or source code available that describes how the Firefox cache actually functions. There is of course the Mozilla source code, but its purpose is not to elucidate cache functionality nor is it concerned with forensic analysis. There are a few other sources of forensically targeted information about the Firefox cache, notably a paper by Symantec and the pyflag source code (see References below), but those sources are incomplete and out-of-date. This paper, in conjunction with the tools developed during the research to write it, will help forensic investigators understand Firefox cache data and assist further research on forensic methodologies in this area.

The paper begins with a general description of the Firefox cache: its location and file structure. I then describe high-level techniques for reading and extracting cache entries. Although forensic techniques are not the primary focus of this paper, I describe some forensic implications to the way Firefox caches data that are called out in the next section, pertaining particularly to acquisition of the Firefox cache. At the end I reach some conclusions and point out areas for future research. And finally, the appendix contains a description of how to use the ff_cache_find tool that was developed to implement the cache extraction techniques described in this paper.

I chose not to include in-depth documentation of the nitty gritty details of the Firefox cache file structure in this paper, but have thoroughly documented these details on my Firefox Forensics Google Code page, available here:https://code.google.com/p/firefox-cache-forensics/wiki/FfFormat. Viewers interested in the file formats and algorithms to process the cache should consult that page.

Note that this paper only covers Mozilla's "Online" cache structure, not the Offline Cache that has a different structure. Also, the formats, tools and techniques described in this paper are valid for all versions of Firefox up to the current version, 10.0.2 as of the time of writing.

Firefox Cache General Description

The Firefox online cache consists of two types of information: "metadata," or information about the cache entries, and "data," which are the actual cached items. I'll describe each in turn.

Firefox cache metadata consists of the following pieces of basic information:

- The complete request string – this is the unique entry to identify a cached item.[1]
- Time/date of cache item first request
- Time/date of the last time the cache item was requested
- Time/date when the cache item will expire
- The "fetch count," or number of times the cache item has been requested
- The complete server response header to the request

Cache data is the complete file as it was returned from the responding server.

The Firefox cache is located in a preconfigured directory under each user profile. The directory location is dependent upon the workstation operating system and version:

Operating System	**Directory Location** (XXXXXXXX is a random alpha-numeric profile identifier)
Windows XP	C:\Documents and Settings\[user]\Local Settings\Application Data\Mozilla\Firefox\Profiles\XXXXXXXX.default\Cache\
Windows Vista and Windows 7	C:\Users\[user]\AppData\Local\Mozilla\Firefox\Profiles\XXXXXXXX.default\Cache\
Mac OS X	~/Library/Caches/Firefox/Profiles/XXXXXXXX.default/Cache *(e.g /Users/[user]/Library/Caches/Firefox/Profiles/XXXXXXXX.default/Cache)*
Linux	~/.mozilla/firefox/XXXXXXXX.default/Cache *(e.g. /home/[user]/.mozilla/firefox/XXXXXXXX.default/Cache)*

Under the Firefox cache directory, the cache consists of an index file, three "internal" cache files and a number of "external" files with hexadecimal names (or, in the case of Mozilla 2.0[2], subdirectories of 0-9,A-F with further subdirectories containing the actual external files). The four files that should always be in the main cache directory are:

- _CACHE_MAP_ – This is the Cache Index file that contains an index to both the metadata and the data cache item and ties the cache metadata to the actual cached data
- _CACHE_001_ – Cache file with 512-byte blocks to store small metadata and data items
- _CACHE_002_ – Cache file with 1024-byte blocks to store medium-sized metadata and data items
- _CACHE_003_ – Cache file with 4096-byte blocks to store large metadata and data items

In addition to these four files, there may be a number of "external" cache files with hexadecimal names, e.g. "C72F9F2Ad01" or "D5517FC3m01". These files store very large data or metadata items. In Mozilla 2.0, these external files are segregated in a subdirectory structure under the Firefox cache directory so that, in the above example, the external files would be named "C/72/F9F2Ad01" and "D/55/17FC3m01." The file naming convention indicates whether the external cache file is for data ("d" in "C72F9F2Ad01") or for metadata ("m" in "D5517FC3m01").

As implied above, Firefox stores smallest data and metadata entries in the _CACHE_001_ file, larger entries in the _CACHE_002_ file, larger still in _CACHE_003_ file, and all the rest in separate external cache files. The algorithm to determine which size cache entry ends up in which file changes slightly between Firefox versions (and can be determined exactly by reading the code) but it appears to be optimizing I/O operations and minimizing zero-padded cache blocks in each file. The _CACHE_001_ file typically contains the most metadata and the _CACHE_003_ file the least, probably because URL request strings and their corresponding server response headers are rarely large enough to be stored in the large-entry cache file.

Reading the Firefox Cache

The process of reading the Firefox cache starts with reading and interpreting each entry in the Cache Mapfile. Entries in the Mapfile indicate which file each of the metadata and the cache data is stored in. If either the metadata or the data is stored in one of the _CACHE_00[1-3]_ files, the Mapfile will indicate the starting block number and the block count, otherwise the Mapfile will give the file name of the external file. In cases where there is no data associated with a metadata entry, the Mapfile entry for the data file won't point to a valid file. The details of how to interpret the Mapfile and _CACHE_00[1-3]_ files are described on the Firefox Cache Format page (https://code.google.com/p/firefox-cache-forensics/wiki/FfFormat).

In addition to pointers to the data and metadata files the Mapfile contains other information helpful for reading the cache. The Mapfile contains a "dirty flag," indicating whether or not the cache had been flushed or not – an indication that the cache may be corrupt. It also contains a version number that's useful for understanding format differences in the cache files.

If one has only the _CACHE_00[1-3]_ files but not a _CACHE_MAP_ file, one can still recover cache metadata and/or cache data but not correlate the two with each other. This is the approach I took for the ff_cache.pm module for log2timeline; given the cache files but no Mapfile, ff_cache.pm reads through the file and determines whether any given block belongs to a metadata entry. If it's metadata, it can be read and integrated into the forensic timeline. Reading a cache file to extract cached data would be a similar exercise in differentiating data from metadata and extracting it. Without the Mapfile, one wouldn't have the original filename or file size information and would need to manually determine and remove NULL-byte padding from the end of an extracted data file – the content-length of a given cached data piece is stored in the metadata information and wouldn't be available unless one can associate metadata with data elements.

Extracting the Firefox Cache

The process to fully extract cached data from the Firefox consists of the following steps:

1. Extract metadata
2. Analyze metadata for information about the cache data (e.g. Content-Length, Content-Type, Content-Encoding, original filename)
3. Extract data associated with metadata
4. Process data (e.g. unzip, rename)

Given the filename, block offset, block size and number of blocks from the Mapfile, extracting the metadata is just a matter of reading fixed-length data from the appropriate location in the appropriate file. Once the metadata segment has been read, the individual fields can be parsed and prepared for analysis. The metadata segment format is described in detail on the Firefox Cache Format page, referenced already and in References, below.

From a forensic point of view, extracting the First Fetch Time and Last Fetch Time data elements are a critical part of parsing the metadata information. These data elements allow creation of a forensic timeline using the cache. This is what the ff_cache.pm log2timeline module does.

Metadata parsing can include digging information out of the Server Response that will be useful for extracting and processing the actual cache data. This can include "Content-Length" HTTP responses to get the byte count of the cached data, "Content-Type" to determine the MIME type of the returned content, and "Content-Encoding" to determine if the returned content has been gzipped. The original filename of the cached item can often be determined by parsing the Request String, and the First or Last Fetch Time can be used to recreate the file

modification time.

Given the filename, block offset, block size, and either number of blocks or the Content-Length information, the Cache Data can be read from the appropriate cache file. The discovered metadata information can be used to recreate the cached file outside of the Firefox cache with its original attributes.

I've written a utility in Perl that uses this methodology to read and extract from a Firefox cache and I've made it available for download. Refer to the appendix for a description of the tool and where to get it.

Forensic Implications

Although I won't discuss all of the methods and implications of doing forensics on Firefox activity (see Alex Bond's excellent blog article for more information about general Firefox forensics), there are implications to forensic methodology based on how information is recovered from the Firefox cache.

While Firefox is running it keeps the working copy of the _CACHE_MAP_ in memory, clearing the contents of the disk version and setting the header "dirty flag." When Firefox flushes the cache, for example upon shutdown, it clears the dirty flag once it has written the contents of the Cache Map file. This is significant to the forensic investigator because Cache Map entries may be corrupt or nonexistent if cache data is acquired while Firefox is either running or has been halted abruptly before flushing its in-memory cache. This indicates that failure to allow Firefox to quit cleanly before acquisition may hinder efforts to do full cache recovery.

Firefox updates the _CACHE_00[1-3]_ cache files continuously during browsing activity (one can observe modification times change on the files and find current entries even while Firefox is still open). These files can be acquired at any time, but without a correct Cache Map file it is difficult to correlate cache metadata with the data it belongs to. For tools that examine only the metadata without attempting to correlate cache metadata and data entries, such as log2timeline, the lack of a valid _CACHE_MAP_ file will not be a handicap.

Because Firefox routinely creates and deletes Cache Mapfiles, it should be possible to forensically recover deleted copies and possibly correlate them to existing and/or recovered deleted _CACHE_00[1-3]_ files. More research is needed in this area.

Conclusion

This paper describes the Firefox cache, including how to read cache metadata and extract cache elements. It points to a technical description of algorithms and file formats for deciphering the Firefox cache, and it presents information on how Firefox manipulates the cache.

Although the paper describes some forensic implications to the way Firefox performs caching, this discussion has not been primarily focused on forensic issues. This paper is meant as a building block for future development to identify and recover deleted Firefox cache elements, correlate Firefox cache findings with other Firefox artifacts such as browse history, and to perform other useful forensic tasks.

Resources

Here are pointers to other resources on this topic that I've put online, as well as sources I've used to write this paper or have found useful to this topic.

- Firefox Cache Format – detailed, in-depth file format and layout for Firefox cache files:https://code.google.com/p/firefox-cache-forensics/wiki/FfFormat

- ff_cache_find.pl – Command-line tool to browse, search and extract Firefox cache entries:https://code.google.com/p/firefox-cache-forensics/downloads/list
- Mozilla Source Code – several versions: http://mxr.mozilla.org/ – root directory with several branches. I used Mozilla Central, Mozilla 1.9.1, 1.9.2 and 2.0.
- "Web Browser Forensics, Part 2" by Keith J. Jones and Rohyt Belani of Symantec description –http://www.symantec.com/connect/articles/web-browser-forensics-part-2. This article is not current to Mozilla 2.0.
- Pyflag – the Forensic and Log Analysis GUI written in python, specifically the Mozilla code:http://www.pyflag.net/pyflag/src/FileFormats/MozCache.py.
- "Firefox 4 Browser Forensics" by Alex Bond "A Renaissance Security Professional"http://renaissancesecurity.blogspot.com/search/label/firefox%204
- "Firefox Forensics" by David Koepi: http://davidkoepi.wordpress.com/2010/11/27/firefoxforensics/
- Kristinn Gudjonsson's Log2timeline tool and source code: http://log2timeline.net/

Appendix: The ff_cache_find tool

As part of research of the Firefox cache, I developed a tool to selectively display and recover cache entries. The tool, ff_cache_find, is a command-line Perl script that reads the Firefox _CACHE_MAP_ file and associated cache files, then displays cache entries metadata and optionally allows recovery of cached items into a recovery directory. This is a description of the tool and how to use it.

ff_cache_find.pl requires the following Perl modules to be loaded on the system:

- Compress::Raw::Zlib – to decompress gzipped content
- MIME::Types – to interpret Content-Type headers

Running the tool with no parameters gives a "usage" message that tells most of the story:

```
Usage: ./ff_cache_find_0.3.pl [mapfile]
Options:
   --search=[search regex] (default is .* - everything)
      Searches against query string and server return information, only returning cache entries matching regex.
   --recover=[recovery dir] - directory to recover files into
      Recommend recovering into a separate empty directory.
      Recovered items will be date-ordered by cache entry modification dates.
```

The mapfile parameter is mandatory and be the pathname of a _CACHE_MAP_ file. If no parameters are provided (other than a _CACHE_MAP_ filename), ff_cache_find will dump a listing of metadata for all cache entries to the screen – for best results, pipe it to a file viewer such as "less."

Providing a regex using the "–search=[regex]" will limit the returned cache entries to those either whose "query" or "server return" string matches the regex. A typical use for this would be to match only cache items from a specific domain, but it can also be used to return cache entries matching anything else in the query or server return: e.g. a "Content-Type" of "image/jpeg".

The "–recover" parameter controls recovery of the cache files into the mandatory named existing directory. It is recommended that the directory be empty in order to simplify analysis of the cache entries. Given the –recover parameter, ff_cache_find will display each cache metadata entry (that matches the optional "–search" regex) and prompt for the desired next action:

```
Do you want to recover this cache item?
[Y]es, [N]o, ne[V]er, [A]lways, [Q]uit (default N):
```

- **Y** will recover the cache entry and display the next cache entry's metadata.
- **N** or ENTER will skip to the next metadata.
- **V** answers "No" to the remaining prompts and displays all remaining metadata entries without recovering them.
- **A** answers "Yes" to the remaining prompts and recovers them without further prompting.
- **Q** exits the script without recovering the current cache entry.

For best results, *do not* pipe the output to "less" when using the –recover option.

Recovered cache entries will be written into the named recovery directory. Ff_cache_find attempts to determine: the cache entry's original filename from the request string, what file type it is, and what its appropriate file extension should be. It sanitizes and truncates the filename and creates a file with the original name and extension, using "[no name]" if that cannot be determined. Each filename has a numeric counter appended to it to avoid overwriting duplicate filenames, e.g. "blank[0].jpg", "blank[1].jpg", etc. Zip-compressed content (as determined by the "Content-Encoding" server return string) will be decompressed into its original size.

In addition to the recovered cache file, ff_cache_find writes a metadata file with the same name but with "_metadata" appended to it. The metadata file contains the cache entry metadata and a description of which cache file the entry was recovered from.

The file modification time of both the recovered cache entry and the accompanying metadata file are set to the time and date of the cache entry's last-modification time in order to retain the original sort order of the cache entries.

The ff_cache_find tool is published under the GPL3, works on all versions of Firefox (as of Firefox 10.0.2) and is available for download at https://code.google.com/p/firefox-cache-forensics/downloads/list

[1] Request strings are normally HTTP requests but may also be for any valid URI type, e.g. FTP or Firefox's custom "wyciwyg" ("What you cache is what you get") URI.

[2] I'm confused by Mozilla's version numbering. In Firefox up to and including v. 3.x, the internal version numbering inside cache files seemed to reflect the Firefox version – e.g. FF2.x had a different internal version number from 3.4, which was different from 3.5, etc., but the actual format of the cache files did not change. As of Firefox 4 and above (up to Firefox 10.0.2 as this was written), the internal version numbering has not changed, so FF6 is the same as FF9 or FF4. Possibly reflecting this change, the cache file format and structure has changed quite a bit between FF<4 and FF>=4. Borrowing from Mozilla's open source code tree structure I've elected to call this latest format "Mozilla 2.0." See the Mozilla versioning table in the linked Firefox Cache Format document for a cross-reference between Firefox and Mozilla version numbering.