

TZWorks® Mozilla Cache Parser (*mcp*) Users Guide



Abstract

mcp is a standalone, command-line tool that parses cache files associated with the Mozilla Firefox Browser. The tool can target various Browser instances of cache and either report the results in a CSV type format or archive the results in a SQLite database. This tool has working versions for Windows, Linux and OS-X.

Copyright © TZWorks LLC

www.tzworks.com

Contact Info: info@tzworks.com

Document applies to v0.21 of ***mcp***

Updated: Apr 15, 2024

Table of Contents

1	Introduction	2
1.1	Location of the Cache data	3
2	How to Use <i>mcp</i>	3
2.1	Targeting Specific Cache files	4
2.2	Processing Cache Files in one or more Subdirectories	6
2.2.1	Parsing Multiple Mozilla Accounts and/or Instances.....	7
2.2.2	Archiving the Content Data.....	8
2.2.3	Splitting the Mozilla Sessions into Separate Files	9
3	CSV Field Names / Meaning.....	10
4	Limitations.....	11
5	Available Options	11
6	Authentication and the License File.....	12
7	References	13

TZWorks® Mozilla Cache Parser (*mcp*) Users Guide

Copyright © TZWorks LLC

Webpage: http://www.tzworks.com/prototype_page.php?proto_id=50

Contact Information: info@tzworks.com

1 Introduction

The Mozilla Cache Parser (*mcp*) targets the Mozilla Firefox cache and extracts useful information for the examiner. This tool is not unique, in that there are other Mozilla cache parsers available; a few good ones are even free. This tool was primarily created based on a need to provide more insight into the association of the cache metadata (eg. timestamps, URL, http request/response, etc) and cache content data (eg. data for the webpage that is displayed), especially when applied to the earlier versions of the Mozilla formats. In addition, and from a tool developer's standpoint, the *mcp* codebase can be used as framework for future work to evaluate Mozilla cache artifact data that may be corrupted or fragmented.

As background, the Mozilla cache, like any other browser cache, is a repository for web data a user has viewed or downloaded. In general, the purpose of the cache is to store data locally, to allow the browser quick access for later requests to that same website. The cache includes: website pages, files, and images that were viewed by a user. In addition to the raw data that was received from a web server, the Mozilla cache also contains useful metadata associated with each item. From the point of view of the forensic examiner the data is interesting, since it contains items such as: the URL of the webpage, number of times the page was fetched from the cache, filename/type/size, last modified time, last fetched time, server time, etc. Having a tool available that can take advantage of this artifact data is necessary to have insights into the user's activity.

Like any other application that stood the test of time, the Mozilla architecture (including cache structure) has evolved over the years. The older version of the Mozilla cache architecture, consisted of 3 categories of files:

1. The `_CACHE_MAP_` which associates the metadata and raw data locations.
2. The `_CACHE_001_`, `_CACHE_002_` and `_CACHE_003_` block files. These 3 files contain predefined chunks (ranging from small chunks to larger chunks) that are used to store the metadata for the cache as well as some raw data from webpages.
3. The last category are the data files (and metadata) that are too large to fit within one of the three block files listed in (2) above. If all the files are available, *mcp* will look at the data in all the files to generate the results. Using the data in the `_CACHE_MAP_`, *mcp* will annotate the location of the raw cache data in the results. In the absence of the `_CACHE_MAP_` file, this associations will not be present.

The newer version of the Mozilla cache uses a separate file per webpage to store both the raw data and the metadata associated with the webpage.

This section's purpose is just a surface discussion on the cache formats primarily to set the stage to let the user know that not all Mozilla cache data is structured the same. Fortunately, the *mcp* the tool can handle the above various format nuances without using any special parameters. It was designed to sense which parsing engine to use and internally adjust the algorithm to appropriately parse any of the cache formats.

1.1 Location of the Cache data

Mozilla cache artifacts are located in the user's directory. This varies depending on the operating system used. Below is a table that breaks out the location by operating system.

OS	Cache location
Win XP	%userprofile%\Local Settings\Application Data\Mozilla\Firefox\Profiles\<random text>.default\Cache
Post Win XP	%userprofile%\AppData\Local\Mozilla\Firefox\Profiles\<random text>.default\Cache
OSX	/Users/[user acct]/Library/Caches/Firefox/Profiles/<random text>.default/Cache
Linux	/home/[user acct]/.mozilla/firefox/<random text>.default/Cache

2 How to Use *mcp*

The screenshot below shows the options available. The formatting options are similar to the rest of the TZWorks tools. The output can be rendered in either: delimited text (CSV and Log2Timeline) or SQLite. The SQLite option was added primarily to allow one to parse the cache records while archiving the results along with any companion content data. More on these options will be discussed later in this document.

```
Administrator: Windows PowerShell

Usage

mcp -file <cache file(s)> [-mapfile <_CACHE_MAP_ file>] [options]
mcp -enumdir <folder> -num_subdirs <# levels> [options]
dir <folder> /b /s /a | mcp -pipe

Format Options
-csv                        = output is CSV format
-csv12t                    = log2timeline output
-dateformat mm/dd/yyyy    = "yyyy-mm-dd" is the default
-timeformat hh:mm:ss      = "hh:mm:ss.xxx" is the default
-no_whitespace            = remove whitespace between delimiter
-csv_separator "|"        = change delimiter to a pipe char
-base10                   = use base10 numbers

File Output Options (some format opts above do not apply to SQLite)
-sqlite <output db>       = *** create/put results in SQLite db
-out <output file>        = put results in text delimited file

Folder Traversing Options
-pipe                     = pipe files to parse
-enumdir <dir> -num_subdirs <#> = pull from files from folder
-split_sessions           = *** Split sessions into separate files
```

To process cache files, one can either target a folder or individual cache files. The tool will automatically determine which version of the format the cache files are in and adjust the parsing engine accordingly. In fact, when parsing many subdirectories of artifacts where each subdirectory is a different account or machine, the tool will dynamically adjust for the version of the cache being parsed at that time and keep the mapping of the cache metadata to the cache content data sorted.

If processing a directory of cache files (either by using the **-pipe** command or the **-enumdir** command), the tool will look for the Mozilla directory structure starting with the "Cache" or "Cache2" folder to indicate when to start parsing. Alternatively, if targeting the older cache format, where you only have a few files, one can use the option **-file <cache file1 | cache file 2 | ...>** as well.

2.1 Targeting Specific Cache files

As mentioned above, if one wants to target a specific cache file or a couple of cache files, one can use the **-file** option. This was included in the options since it was needed during the debugging of the **mcp** tool. With this option, it is useful to analyze one file at a time to help debug what is going on within the parsing engine. In the example below, we are targeting a cache file that uses the older version of the cache format (eg **_CACHE_001_**). The results would be rendered in the test1.csv file.

```
>mcp64 -file .\cache_test1\_CACHE_001_ -out test1.csv
```

Pipe delimited text is the default output that is rendered by the tool. On the left is the Mozilla version of the cache format (major and minor version separated by a dot). Many of the other fields are the metadata associated with Firefox requesting a page/data and the server serving up the webpage/data. Shown in the screenshot is only the server timestamp, but also in the data (truncated in the screenshot

below), is up to 6 – 8 timestamps that range from those associated with the server, browser and file timestamps.

cache_version	request_type_reply_status	serv_name	serv_timezone	serv_date	fetch_count	url
cache firefox [1.19]	("request_type":"POST";"reply_status":"http/1.1 200 ok")	ECS (mic/9A5	GMT	05/16/2020 15:01:38	05/	1 id=Sec00001&uri=http://ocsp.digicert.com/
cache firefox [1.19]	("request_type":"GET";"reply_status":"http/1.1 200 ok")	sffe	GMT	05/16/2020 15:00:51	0	2 https://www.google.com/images/searchbo
cache firefox [1.19]	("request_type":"GET";"reply_status":"http/1.1 200 ok")	gws	GMT	05/16/2020 15:00:51	1	https://www.google.com/search?hl=en&sou
cache firefox [1.19]	("request_type":"GET";"reply_status":"http/1.1 204 no content")	gws	GMT	05/16/2020 15:00:51	1	https://www.google.com/gen_204?s=web&
cache firefox [1.19]	("request_type":"GET";"reply_status":"http/1.1 200 ok")	ESF	GMT	05/16/2020 15:01:13	1	https://fonts.googleapis.com/css?family=Op
Cache type/version	st_type":"GET";"reply_status":"http/1.1 301 moved permanently")	Apache/2.4.6	GMT	05/16/2020 15:00:12	1	https://www.mozilla.com/about/
cache firefox [1.19]	st_type":"GET";"reply_status":"http/1.1 301 moved permanently")	cloudflare	GMT	05/16/2020 15:01:38	1	https://www.mozilla.org/firefox/about/
cache firefox [1.19]	("request_type":"GET";"reply_status":"http/1.1 200 ok")	cloudflare	GMT	05/16/2020 15:01:38	1	https://www.mozilla.org/en-US/about/

content_type	content_filename	content_size	cntdata_location_info	extra_fields
application/ocsp-response		471	("src":"mapfile";"location":"_CACHE_001_";"offset":"0x7e00")	{server_response:["accept-ranges":"bytes"],"age
image/png	desktop_se	665	("src":"mapfile";"location":"_CACHE_001_";"offset":"0x6000")	{server_response:["accept-ranges":"bytes"],"alt
text/html; charset=UTF-8	search	50541	("src":"mapfile";"location":"B/A2/E4D38d01";"context_hint"	{server_response:["alt-svc":"h3-27=:443";"ma=259
text/html; charset=UTF-8	gen_204	0		{server_response:["alt-svc":"h3-27=:443";"ma=25
text/css; charset=utf-8	css	240	("src":"mapfile";"location":"_CACHE_001_";"offset":"0x5f00")	{server_response:["accept-ranges":"bytes"],"age
text/html; charset=iso-8859-1		245	("src":"mapfile";"locat	{server_respo
text/html; charset=utf-8	about	0		{server_respo
text/html; charset=utf-8		96069	("src":"mapfile";"location":"9/5B/C07A6d01";"context_hint"	{server_response:["accept-ranges":"bytes"],"cac

One should note that even though there is no `_CACHE_MAP_` file being used in the above example, the **mcp** tool will still try to parse out where the “content data” is located. For the older Mozilla cache formats (such as in this example), the association between the ‘metadata’ and ‘content data’ is recorded in the `_CACHE_MAP_` file, so without this *mapping file* being present, the **mcp** tool needs to use its internal scanning to try to locate ‘content data’. For this example, content data was located within the `_CACHE_001_` file itself. This content data is then attempted to be matched with the metadata parsed.

On the field labelled ‘`cntdata_location_info`’ in the above output, this is where **mcp** will put the guesses between the metadata and content data associations. Therefore, if the output uses the notation “*rawscan*”, then the tool is telling the user, this result is a product of the tools’ heuristics of comparison and it deems it a match. The heuristics use a number of tests, including: size of the content data, content data type, etc.

If one inspects the cache file at the offset suggested by the heuristics, one can verify whether this match has some confidence of being correct, or whether it is just a ‘false positive’. Below is a hex dump for the first entry above (offset 0x7e00). Those familiar with X509 certificates will recognize this is what the data is, and it matches the *content-type* in the metadata which was expecting an “OCSP-Response”.

_CACHE_001_ x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
7E00h:	30	82	01	D3	0A	01	00	A0	82	01	CC	30	82	01	C8	06	0..ô... .î.ô.ê.
7E10h:	09	2B	06	01	05	05	07	30	01	01	04	82	01	B9	30	82	.+.....0.....'0.
7E20h:	01	B5	30	81	9E	A2	16	04	14	03	DE	50	35	56	D1	4C	.µ0.žć... .pP5VñL
7E30h:	BB	66	F0	A3	E2	1B	1B	C3	97	B2	3D	D1	55	18	0F	32	»fðÊâ... .Ä-²=ñU...2
7E40h:	30	32	30	30	35	31	35	31	37	32	38	31	35	5A	30	73	0200515172815Z0s
7E50h:	30	71	30	49	30	09	06	05	2B	0E	03	02	1A	05	00	04	0q0I0...+.....
7E60h:	14	80	51	06	01	32	AD	9A	C2	7D	51	87	A0	E8	87	FB	.€Q..2-šÄ}Q‡ è‡û
7E70h:	01	62	01	55	EE	04	14	03	DE	50	35	56	D1	4C	BB	66	.b.Uî... .pP5VñL»f

Going another level deeper, if one adds the `_CACHE_MAP_` file into the mix during the parse option, the command will look like this.

```
>mcp64 -file .cache_test1\_CACHE_001_ -mapfile .\cache_test1\_CACHE_MAP_ -out test2.csv
```

Since the *mapfile* was added during the parsing operation, the *mcp* tool will make use of the `_CACHE_MAP_` file to associate all the metadata records to the content data. The companion output is shown below. The output is truncated to just show the right side of the formatted data so one can see how this compares to the previous example. One can see the *mapfile* associations allowing the *mcp* tool to populate the content data with a higher confidence. For the most part most of the entries agree with the heuristics done previously. Finally, one should note, that even though the `_CACHE_003_` file was not parsed, it shows up as a source of the content data, since that was identified with the `_CACHE_MAP_` entry.

content_type	content_filename	content_size	cntdata_location_info	Context data location populated by <code>_CACHE_MAP_</code> data and thus have a higher confidence of being accurate
application/ocsp-response		471	{"src":"mapfile","location":"_CACHE_001_","offset":"0x1000","	
application/ocsp-response	gts1o1	472	{"src":"mapfile","location":"_CACHE_001_","offset":"0x2200","	
application/ocsp-response	gsr2	468	{"src":"mapfile","location":"_CACHE_001_","offset":"0x2600","	
application/ocsp-response		471	{"src":"mapfile","location":"_CACHE_001_","offset":"0x3000","	
application/ocsp-response	gts1o1	472	{"src":"mapfile","location":"_CACHE_001_","offset":"0x3500","	
text/xml; charset=utf-8	rss.xml	5899	{"src":"mapfile","location":"_CACHE_003_","offset":"0x025000"	
text/html; charset=UTF-8		12710	{"src":"mapfile","location":"_CACHE_003_","offset":"0x027000"	

The above discussion focused on the older Mozilla cache format (version 1.x format). The later cache formats overcome this mapping issue by integrating the cache's metadata within the same file as the content data. This eases the parsing logic since only one file needs to be analyzed for both the metadata and content data.

2.2 Processing Cache Files in one or more Subdirectories

If desiring to process many Mozilla cache files in one pass, one can make use of *mcp's* piping option (*-pipe*) or the folder enumeration option (*-enumdir*). Either of these options allow one to target multiple subdirectories during the parsing operation. Below is a simple way to target the cache files in a Mozilla account.

```
>dir C:\Users\tzlabs\AppData\Local\Mozilla\Firefox\Profiles /b /s /a | mcp64 -pipe -out test3.csv
```

If one is uncertain where exactly the Mozilla cache files are located, then the following works as well, but is a little slower since the command will enumerate many other 'non-cache' files. The results, however, should be the same.

```
>dir C:\Users\tzlabs /b /s /s | mcp64 -pipe -out test4.csv
```

If desiring more control on the number of subdirectories to traverse, one can use the **-enumdir** option along with the **-num_subdirs** sub-option. It would look like this for the above example:

```
>mcp64 -enumdir c:\users\tzlab -num_subdirs 10 -out test5.csv
```

For any of the above options to work with this tool, the Mozilla folder structure must be preserved after the random session text string and before the 'Cache' or 'Cache2' folder. Why? It was a design choice to allow the tool to easily tell the 'type' a file the tool was examining to assist in parsing; this assists the tool to determine the version of the format of the cache being used (due to the naming convention). Furthermore, the naming convention also allows the tool whether it should switch modes to handle multiple Mozilla accounts during one session run.

2.2.1 Parsing Multiple Mozilla Accounts and/or Instances

Since **mcp** makes uses of the Mozilla directory structure, one can pass in a number of accounts for the tool to process in one session. Internally, the **mcp** tool will detect the change in Mozilla instance and/or account and flush the current instance/account prior to processing the next instance/account. In this way, the tool to conserves memory usage on the host machine. This is useful if trying to parse many Mozilla cache collections at one time.

When outputting the results, the tool defaults by integrating the output into one file. For testing purposes, this technique allows one to run the tool against many different versions of the Mozilla cache and verify its accuracy and performance. Below is an example of one of the testcase setups that is used internally to test out the tool. (Note: if the reader has a collection of other testcases that they are willing to share, please contact *TZWorks* so they can be added to the current test suite). The collection of data is taken from Mozilla versions 3 to 77.



When running the **mcp** tool against the root folder, in this case “*sqlite\Mozilla*”, it will traverse each of the subfolders and try to target each cache file it finds.

```
>dir e:\sqlite\Mozilla /b /s /a | mcp64 -pipe -out test6.csv
```

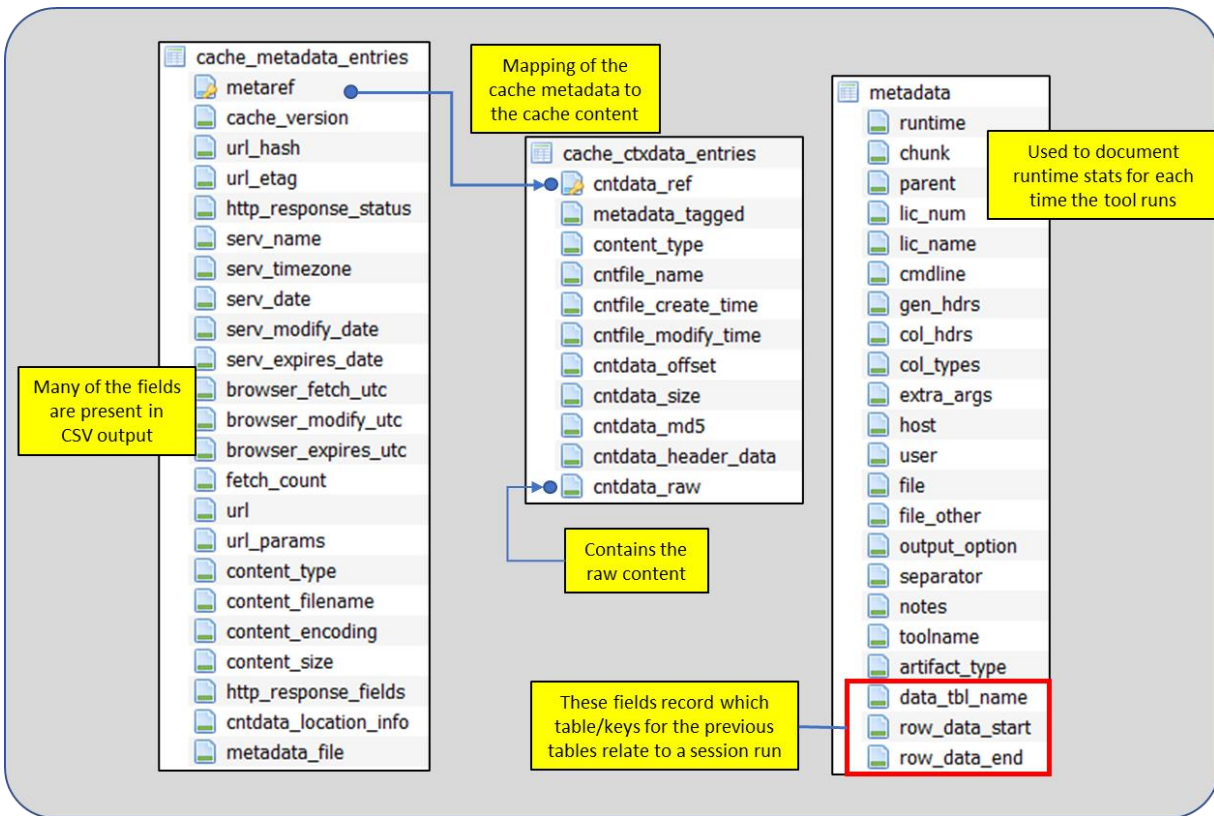
When done, the results will be integrated into a very large results file. The context of the metadata is preserved in the output, since there is a delimited field that includes the source cache path/filename.

2.2.2 Archiving the Content Data

With the default option, the tool sends the parsed output to delimited text. This is fine when only wanting the results associated with the metadata and pointers to the content data. If desiring to archive the content data as well, **mcp** has an option to create and output the results into a SQLite database.

To invoke this option, use the **-sqlite <db_name>** in your command. All parsed results will include both the record metadata and its associated content data. To view the results, one will need to be familiar with the SQL syntax to query the database, or alternatively, will need a separate SQLite viewer to look at the data. A good SQLite viewer is the “DB Browser for SQLite” and a reference is located at the end of this document.

The database schema created by **mcp** consists of 4 tables: (a) *cache_metadata_entries*, (b) *cache_ctxdata_entries*, (c) *metadata* and (d) *ref*. Only the first two tables have the records from the parsed metadata and content data, respectively. The *metadata* table is used to record the session parameters used when running the parser. The last table (*ref*), is not shown in the diagram, and is used internally by **mcp** for bookkeeping only. The fields for the first three tables and their relationship are shown below.



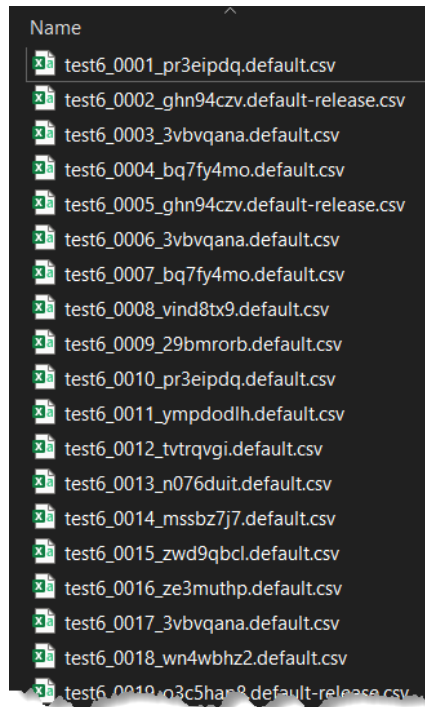
The records in the `cache_metadata_entries` table are similar to the information rendered in the CSV output. The actual content data is stored in the `cache_ctxdata_entries` table under the field name "ctxdata_raw". This is a 'blob' type since the data can be either text or binary.

2.2.3 Splitting the Mozilla Sessions into Separate Files

One can take the discussion in the previous sections and modify the output so that the data is broken out into separate files per Mozilla session. This applies to both the CSV and the SQLite output variants. The syntax is the same as before, however, one just appends the sub-option **-split_sessions** to the command. This tells the **mcp** tool to take whatever was specified as the output file to be appended with a session number along with the random string used by the Mozilla folder name. Below is an example using this syntax.

```
>dir e:\sqlite\Mozilla /b /s /a | mcp64 -pipe -out test6.csv -split_sessions
```

When the processing is done, one will have a number of files (one per Mozilla session). The output notation will be something like what is shown below. The output name specified (in this case "test6") will be the part of the name with an incremented number along with the folder name used by Mozilla for that session.



3 CSV Field Names / Meaning

Below is a reference of all the CSV fields used and their meanings.

CSV Field	Definition
type	Cache version number
url_hash	SHA1 hash of the URL contained in the metadata. This is a computed value by <i>mcp</i> . This hash should be equivalent to the filename for those cache versions that show a SHA1 hash for the name.
url_etag	The HTTP etag that was present in the HTTP response
request_type_reply_status	HTTP request type (eg. GET, POST), and reply status (eg. HTTP/1.1 200 OK)
serv_name	Server name recorded in the HTTP Response
serv_timezone	Server time zone
serv_date	Server timestamp included in the HTTP Response
serv_modify_date	Server modify timestamp included in the HTTP Response
serv_expires_date	Server expire timestamp included in the HTTP Response
browser_fetch_utc	Browser - last time the cache was fetched
browser_modify_utc	Browser modify timestamp associated with the cache
browser_expires_utc	Browser expire timestamp associated with the cache
content_create_utc	Actual content data file create timestamp. This is only present if the content file is a separate file. For Linux and OSX, this is the status change timestamp
content_modify_utc	Actual content data file modify timestamp. This is only present if the content file is a separate file.

fetch_count	Number of times the cache was fetched
url	URL of the webpage visited
url_params	Any URL parameters used. This is formatted as JSON.
content_type	The content data type (eg. GIF, JPEG, text, etc) extracted from the HTTP response
content_filename	Last part of the URL prior to the URL parameters extracted from the HTTP response
content_encoding	The encoding used on the content data (eg. gzip, br, etc) extracted from the HTTP response
content_size	Size of the content data extracted from the HTTP response
content_location_info	The file and offset (if not zero) within the file where the content data is located. This is formatted as JSON.
extra_fields	The key/value pairs extracted from the HTTP response. This is formatted as JSON.
file	The original path/file containing the metadata

4 Limitations

This version of the tool has a number of limitations. They are listed below.

- The tool is still prototype in nature being that this is the first version released. It still needs to be tested against various types of files, corrupted files, etc. to ensure the tool can perform consistently.
- The earliest version of the Mozilla cache this tool has been tested on is v3.0.1. Therefore, prior versions may or may not work; and if they seem to work, may or may not yield accurate results.
- The folder enumeration of the cache file option relies on the Mozilla directory structure as well as the naming convention used by Mozilla. Therefore, if either of these things are changed by Mozilla or if changed by a user, the parsing engine will have unpredictable results or no results at all.
- There are a couple of parsing engines within this tool; which engine is used is a function of the Mozilla naming convention used for the cache file.

5 Available Options

Option	Description
-csv	Outputs the data fields delimited by commas. Since filenames can have commas, to ensure the fields are uniquely separated, any commas in the filenames get converted to spaces.
-csvl2t	Outputs the data fields in accordance with the log2timeline format.

-sqlite	Outputs the data into a SQLite database. The syntax is: -sqlite <db name to create or use> .
-pipe	Used to pipe files into the tool via STDIN (standard input). Each file passed in is parsed in sequence.
-enumdir	Experimental. Used to process files within a folder and/or subfolders. Each file is parsed in sequence. The syntax is -enumdir <folder> -num_subdirs <#> .
-filter	Filters data passed in via STDIN via the -pipe option. The syntax is -filter <"*.ext *partialname* ..."> . The wildcard character '*' is restricted to either before the name or after the name.
-no_whitespace	Only applies to -csv and -csvl2t options. Used in conjunction with -csv option to remove any whitespace between the field value and the CSV separator.
-csv_separator	Only applies to -csv and -csvl2t options. Used in conjunction with the -csv option to change the CSV separator from the default comma to something else. Syntax is -csv_separator "/" to change the CSV separator to the pipe character. To use the tab as a separator, one can use the -csv_separator "tab" OR -csv_separator "\t" options.
-dateformat	Output the date using the specified format. Default behavior is -dateformat "yyyy-mm-dd" . Using this option allows one to adjust the format to mm/dd/yy, dd/mm/yy, etc. The restriction with this option is the forward slash (/) or dash (-) symbol needs to separate month, day and year and the month is in digit (1-12) form versus abbreviated name form.
-timeformat	Output the time using the specified format. Default behavior is -timeformat "hh:mm:ss.xxx" . One can adjust the format to microseconds, via "hh:mm:ss.xxxxxx" or nanoseconds, via "hh:mm:ss.xxxxxxxxxx" , or no fractional seconds, via "hh:mm:ss" . The restrictions with this option is a colon (:) symbol needs to separate hours, minutes and seconds, a period (.) symbol needs to separate the seconds and fractional seconds, and the repeating symbol 'x' is used to represent number of fractional seconds.
-quiet	Show no progress during the parsing operation.
-split_sessions	Split the Mozilla sessions into separate files.
-utf8_bom	All output is in Unicode UTF-8 format. If desired, one can prefix an UTF-8 <i>byte order mark</i> to the output using this option.

6 Authentication and the License File

This tool has authentication built into the binary. The primary authentication mechanism is the digital X509 code signing certificate embedded into the binary (Windows and macOS).

The other mechanism is the runtime authentication, which applies to all the versions of the tools (Windows, Linux and macOS). The runtime authentication ensures that the tool has a valid license. The license needs to be in the same directory of the tool for it to authenticate. Furthermore, any modification to the license, either to its name or contents, will invalidate the license.

7 References

1. Mozilla-central [<https://hg.mozilla.org/mozilla-central/annotate/80eff2b52d14/network/cache2/CacheFileMetadata.h#l54>]
2. Mozilla-central [<https://dxr.mozilla.org/mozilla-central/source/network/cache2/CacheIndex.h>]
3. firefox-cache-forensics -FfFormat.wiki [<https://code.google.com/archive/p/firefox-cache-forensics/wikis/FfFormat.wiki>]
4. Joachim Metz. Firefox cache file format [<https://github.com/libyal/dtformats/blob/master/documentation/Firefox%20cache%20file%20format.asciidoc>].
5. Endpoint Protection / Web Browser Forensics part 2 [<https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=30e9590f-e848-4857-8bd1-adf70638af36&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments>]
6. SQLite library statically linked into tool [Amalgamation of many separate C source files from SQLite version 3.32.3].
7. SQLite documentation [<http://www.sqlite.org>].
8. DB Browser for SQLite [<http://sqlitebrowser.org/>]