

TZWorks® Mozilla SQLite Parser (*msp*) Users Guide



Abstract

msp is a standalone, command-line tool that parses SQLite files associated with the Mozilla Firefox Browser (or other browsers based on the *Gecko* browser engine, such as SeaMonkey) that is used on desktops. The tool can target various Firefox SQLite databases and report the results in a CSV type format. 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 ***msp***

Updated: Apr 24, 2025

Table of Contents

1	Introduction	3
2	Databases Targeted by this tool	4
2.1	<i>places.sqlite</i> Database	4
2.2	<i>cookies.sqlite</i> Database	6
2.3	<i>downloads.sqlite</i> Database	6
2.4	<i>favicons.sqlite</i> Database	6
2.5	<i>formhistory.sqlite</i> Database	7
2.6	Location of the SQLite databases	7
3	How to Use <i>msp</i>	9
3.1	Targeting Specific Database files	9
3.2	Integrated Parsing Algorithms	10
3.2.1	Algorithms and their Pros/Cons	11
3.3	Modified CSV Output	12
3.4	Type Designations	12
3.5	Processing Multiple Databases	13
3.6	Merging of Data between Tables	14
3.7	Parsing Firefox Artifacts from Memory or a Disk Image	14
3.8	Bypassing the Embedded SQLite library	15
3.9	Splitting the Mozilla Sessions into Separate Files	16
3.10	Verification and Validation	17
4	Use of the SQLite Library	18
5	CSV Field Names / Meaning	18
6	Limitations	19
6.1	Versions of Firefox tested with <i>msp</i>	20
7	Available Options	20
8	Authentication and the License File	22
9	References	22

TZWorks® Mozilla SQLite Parser (*msp*)

Users Guide

Copyright © TZWorks LLC

Webpage: http://www.tzworks.com/prototype_page.php?proto_id=49

Contact Information: info@tzworks.com

1 Introduction

As background, the *Gecko* engine is used in the current *Mozilla* architecture. Other common browser engines include *Blink* and *WebKit*. Below is a table to showing where the *Gecko engine* is used, and consequently, which browsers and the respective SQLite tables, the *msp* tool targets.

Tool	Browser Type	SQLite Tables Targeted by Tool	Notes
<i>csp</i>	Chromium based that use the <i>Blink</i> engine (e.g. <i>Edge</i> , <i>Chrome</i> , <i>Brave</i> , <i>Vivaldi</i> , etc.)	<i>urls</i> , <i>visits</i> , <i>keyword_search_terms</i> , <i>visit_source</i> , <i>downloads</i> , <i>downloads_url_chains</i> , <i>clusters_and_visits</i> , <i>content_annotations</i> , <i>context_annotations</i> , <i>cookies</i> , <i>autofill</i> , <i>thumbnails</i> , <i>top_sites</i> , <i>omni_box_shortcuts</i> , <i>logins</i> , <i>favicons</i> , <i>favicon_bitmaps</i> , <i>nel_policies</i> , <i>bounces</i>	The <i>csp</i> tool just targets the SQLite data and the <i>ccp</i> tool is used to parse the cache
<i>msp</i>	<i>Mozilla</i> based that use the <i>Gecko</i> engine (e.g. <i>Firefox</i> , <i>SeaMonkey</i> , <i>Tor Browser</i> , etc.)	<i>moz_places</i> , <i>moz_Origins</i> , <i>moz_bookmarks</i> , <i>moz_historyvisits</i> , <i>moz_inputhistory</i> , <i>moz_keywords</i> , <i>moz_annos</i> , <i>moz_items_annos</i> , <i>moz_anno_attributes</i> , <i>moz_cookies</i> , <i>moz_downloads</i> , <i>moz_icons</i> , <i>moz_icons_to_pages</i> , <i>moz_pages_w_icons</i> , <i>moz_favicons</i> , <i>moz_formhistory</i>	The <i>msp</i> tool just targets the SQLite data and the <i>mcp</i> tool is used to parse the cache
<i>sap</i>	<i>WebKit</i> based browsers (e.g. <i>Safari</i>)	<i>history_items</i> , <i>history_visits</i> , <i>history_items_to_tags</i> , <i>history_tags</i> , <i>icon_info</i> , <i>page_url</i> , <i>cache_settings</i> , <i>cloud_tabs</i> , <i>cloud_tab_devices</i> , <i>cfurl_cache_blob_data</i> , <i>cfurl_cache_receiver_data</i> , <i>cfurl_cache_response</i> , <i>ItemTable</i>	The <i>sap</i> tool also parses the cache, as well as, some <i>plist</i> s containing useful data

This document will use the terms *Mozilla* and *Firefox*, but the tool's capabilities will also handle the *SeaMonkey* and *Tor* browsers. The *Firefox* version of the browser is usually the most kept up to date as far as latest changes.

As shown in the second row above, the *Mozilla Firefox* Browser has many artifacts available that the forensics examiner can use in identifying a user's Internet activity. This includes *Firefox*'s various databases, local storage, JSON formatted text files and its cache.

This tool only addresses certain SQLite databases and specific tables within those databases that are used by the desktop version of *Firefox*, *SeaMonkey* and *Tor* browsers that have been deemed useful by the forensics community. Specifically, this tool currently targets the following five databases: (a) *places.sqlite*, (b) *cookies.sqlite*, (c) *downloads.sqlite*, (d) *favicons.sqlite*, and (e) *formhistory.sqlite*. Each

of these databases will be discussed later in the document. This tool focuses on the desktop platform of Firefox and not the versions that can be used with iOS and Android.

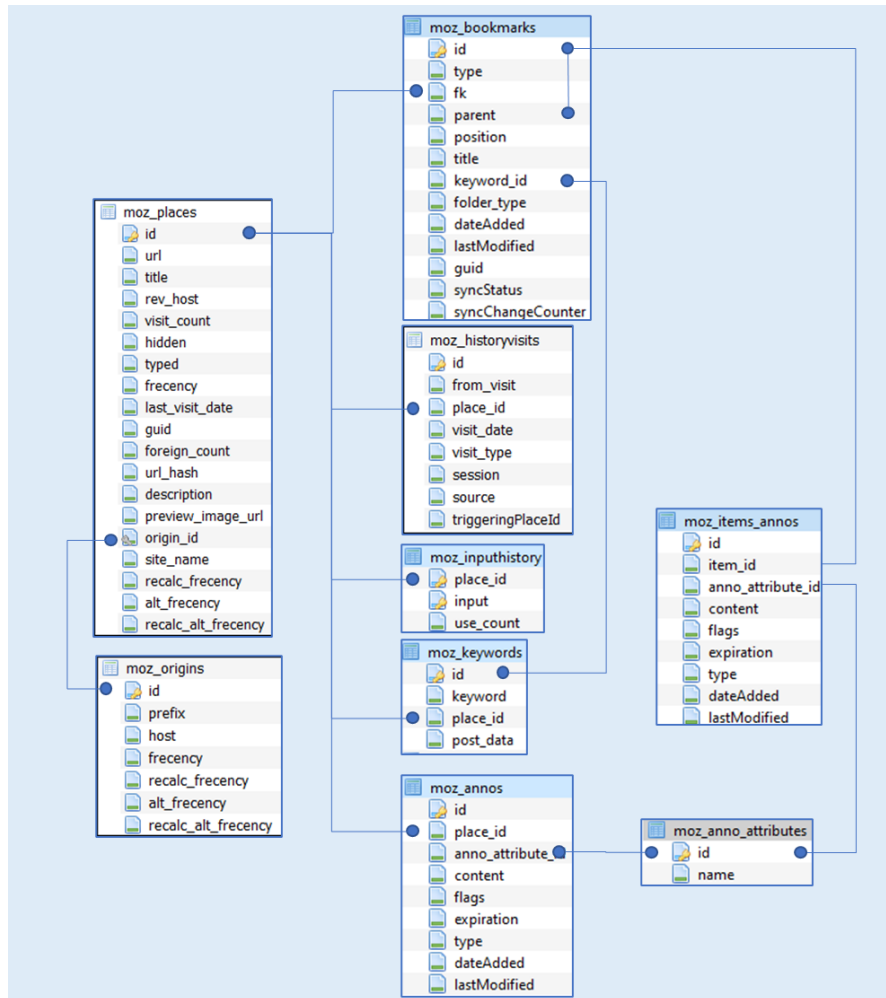
When looking across the various versions of the Firefox Browser over time, the schemas of the databases have evolved. One can think of the database schema as the roadmap that defines the fields and the type of data in each field that comprise a record in the table (where one or more tables reside in a database). The change in schemas across different versions is something that needed to be taken into account when designing the *m_{sp}* tool. The design that was used was similar to that used on past SQLite parsers developed by TZWorks, in that, the tool can dynamically detect and adjust to varying schemas as they are encountered during the parsing operation.

In addition to the auto-schema detection, the *m_{sp}* tool allows the user to parse a target database in three ways. (1) The first way makes use of the standard SQL (Structure Query Language) to parse the records. The SQL syntax is internal to the tool, so the user is not required to have any knowledge about SQL or its syntax. For this option to be available, the SQLite library was statically linked into the tool, which eliminates the need for a SQLite dynamic library to be present to run the tool. (2) The second approach allows the user to instruct the tool to parse each record by traversing the internal SQLite structures as they are encountered. This option does not use any part of the standard SQLite library, but utilizes the TZWorks' internally designed libraries. The benefit of 'rolling your own' library is multi-faceted; not only does it allow the tool to extract records from a corrupted database, but one can annotate the exact offset of the data where it was found. This enables one to easily validate it later with a hex-editor. (3) The third, and final approach, uses a *signature-based* parse. While this option is more limited in merging records from one table to another, this turns out to be a unique way in parsing a blob of data whether it be from memory or from a fragment of a database. All three approaches are designed into the tool for the analyst to use. More discussion on these options is discussed later.

2 Databases Targeted by this tool

2.1 *places.sqlite* Database

Mozilla's *places.sqlite* database has a number of tables of interest to the analyst. Below is a diagram of these tables and their relationships to each other. Keep in mind not all the tables, as well as fields in the tables, may be present in the older Firefox browser versions. The same can be said of the fields that comprise each of the tables. Some fields may not be present depending on your version of Firefox.

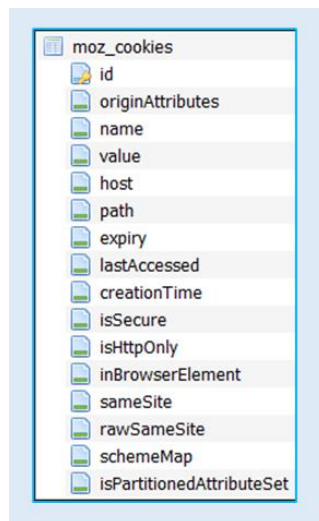


The table relationships are shown by the lines connecting one table to another. These relationships will have an effect on the number of records that will be outputted by the **m_{sp}** tool. For example, the tables *moz_places* and *moz_historyvisits* have what is called a ‘one to many’ relationship. The *moz_historyvisits* may have many linked records to only one entry in the *moz_places* table. Therefore, after merging the data from the *moz_historyvisits* table to the *moz_places* table, one most likely will get more records in the output of the report than the number of records in the *moz_places* table. This is because each parsed line in the output has taken the ‘one to many’ relationship and converted it to a ‘one to one’ relationship; where each line in the output shows one *moz_places* entry and one *moz_historyvisits* entry. If there was a second *moz_historyvisits* entry for the same *moz_places* entry, that would constitute a separate output line. Outputting the data this way allows the various timestamps recorded to be digested better by other tools.

This behavior exists across other tables as well, assuming there are multiple entries from one table referencing a single entry in another table.

2.2 *cookies.sqlite* Database

The *cookies.sqlite* database has one table of interest shown below.

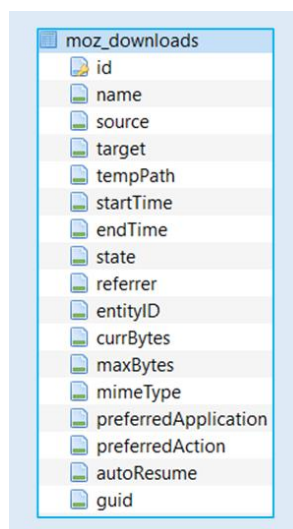


A screenshot of a database viewer showing the schema of the `moz_cookies` table. The table has 16 columns, each with a small icon to its left. The columns are listed in a single column within a light blue bordered box.

moz_cookies
id
originAttributes
name
value
host
path
expiry
lastAccessed
creationTime
isSecure
isHttpOnly
inBrowserElement
sameSite
rawSameSite
schemeMap
isPartitionedAttributeSet

2.3 *downloads.sqlite* Database

The *downloads.sqlite* database is only seen with the older versions of Firefox. The newer versions store the download data within the *places.sqlite* database. The **m^{sp}** tool can extract these records from either database. Shown below is the database that exists with the older version of the Firefox browser.

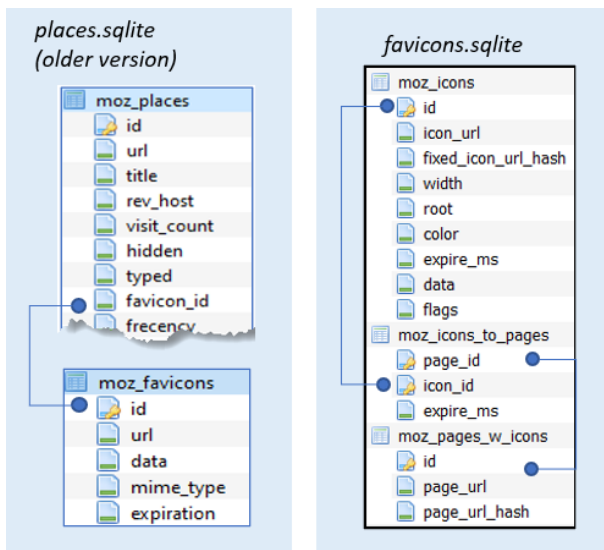


A screenshot of a database viewer showing the schema of the `moz_downloads` table. The table has 18 columns, each with a small icon to its left. The columns are listed in a single column within a light blue bordered box.

moz_downloads
id
name
source
target
tempPath
startTime
endTime
state
referrer
entityID
currBytes
maxBytes
mimeType
preferredApplication
preferredAction
autoResume
guid

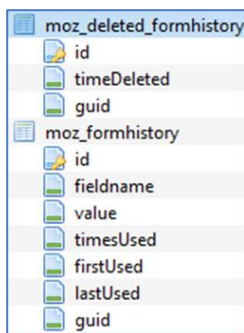
2.4 *favicons.sqlite* Database

The *favicons* are either stored in the *places.sqlite* database under the table *moz_favicons* or in a separate database called *favicons.sqlite*. The older versions of Firefox stored the Favicons in the *places.sqlite* database and the newer versions are stored in a separate database. The **msp** tool can extract these records from either type.



2.5 formhistory.sqlite Database

The *formhistory.sqlite* database has two tables. One table is for deleted *form history* records and the other table for the *form history*. The **msp** tool only extracts records from the *moz_formhistory* table.



2.6 Location of the SQLite databases

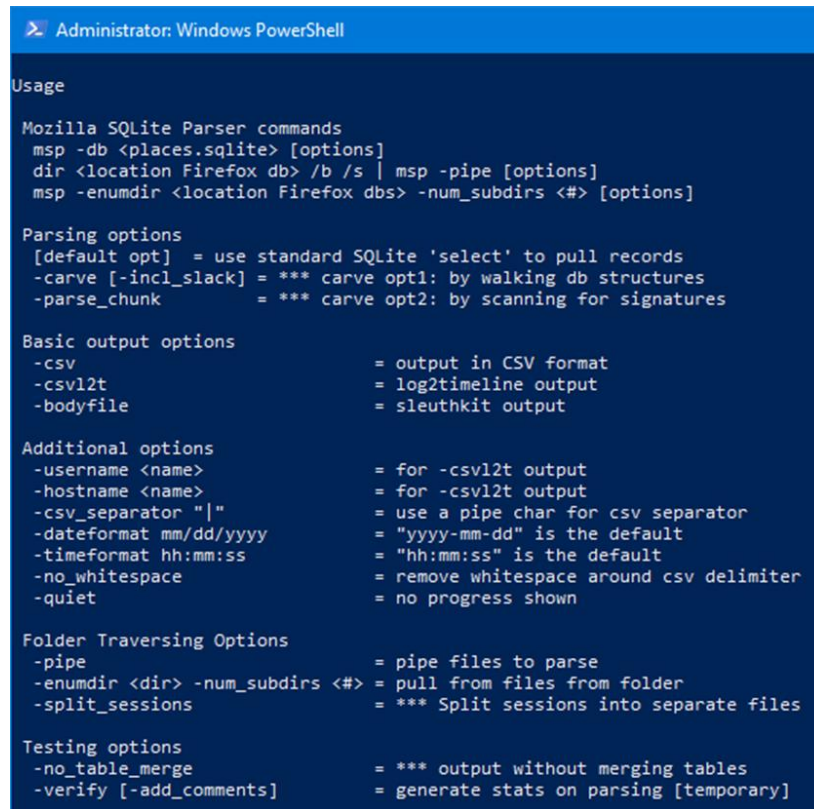
Mozilla Firefox SQLite database artifacts are located in the user's directory. This varies depending on the operating system used. Below is a table that breaks out the location by operating system.

OS	Database location
----	-------------------

Win XP	%userprofile%\Application Data\Mozilla\Firefox\Profiles\<random text>.default
Post Win XP	%userprofile%\AppData\Roaming\Mozilla\Firefox\Profiles\<random text>.default
OSX	/Users/[user acct]/Library/Application Support/Firefox/Profiles/<random text>.default
Linux	/home/[user acct]/.mozilla/firefox/<random text>.default

3 How to Use *mzp*

The screenshot below shows the options available. The output formatting options are similar to the rest of the TZWorks tools. The output can be rendered in one of the three formats: CSV, *Log2Timeline*, or *BodyFile* (*Sleuthkit* format).



```
Administrator: Windows PowerShell

Usage

Mozilla SQLite Parser commands
mzp -db <places.sqlite> [options]
dir <location Firefox db> /b /s | mzp -pipe [options]
mzp -enumdir <location Firefox dbs> -num_subdirs <#> [options]

Parsing options
[default opt] = use standard SQLite 'select' to pull records
-carve [-incl_slack] = *** carve opt1: by walking db structures
-parse_chunk       = *** carve opt2: by scanning for signatures

Basic output options
-csv                = output in CSV format
-csvl2t             = log2timeline output
-bodyfile           = sleuthkit output

Additional options
-username <name>    = for -csvl2t output
-hostname <name>    = for -csvl2t output
-csv_separator "|"  = use a pipe char for csv separator
-dateformat mm/dd/yyyy = "yyyy-mm-dd" is the default
-timeformat hh:mm:ss = "hh:mm:ss" is the default
-no_whitespace      = remove whitespace around csv delimiter
-quiet              = no progress shown

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

Testing options
-no_table_merge     = *** output without merging tables
-verify [-add_comments] = generate stats on parsing [temporary]
```

To process SQLite database files, one can either target a folder or individual database files. The tool will automatically determine which database type/schema version to use 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 database being parsed at that time and keep the record content data sorted.

If processing a directory of database files (either by using the **-pipe** command or the **-enumdir** command), the tool will look for the Mozilla directory structure starting with the "Profile" folder to indicate when to start parsing.

3.1 Targeting Specific Database files

If one wants to target a specific database, use the **-db** option. Without any specific parsing parameters, the default parser uses the Structured Query Language (SQL) in combination with the

statically linked SQLite library to extract the records in the various tables in the database. Below is an example of doing this.

```
>msp64 -db c:\dump\places.sqlite -out results.csv
```

The default output is rendered in pipe delimited text and has 11 fields. These fields are explained in the section on *CSV Field Names/Meaning*. To allow flexibility with rendering differing data types across the differing tables and databases in the output, some of the fields make use of a quasi-JSON like format; this allows records with different fields across various tables to be rendered in one CSV/delimited format. Below is a sample output.

type	rowid	create time	last access	expires	url or name	params	params translation	extra fields	data record	file
url	1	10/16/2019	10/16/2019		https://www.mozilla.org/privacy/firefox/	{"data":"none"}	{"visit_type":"link"}	{historyvisit={"id":"1","places":		E:\testcase\
url	2	10/16/2019	10/16/2019		https://www.mozilla.org/en-US/privacy/firefox/	{"title":"Firefox	{"visit_type":"permanent i	{historyvisit={"id":"2","places		E:\testcase\
url	3	05/14/2020	05/14/2020		https://www.mozilla.org/en-US/firefox/76.0.1/w	{"oldversion":"	{"visit_type":"link"}	{historyvisit={"id":"3","places		E:\testcase\
url	4	05/14/2020	05/14/2020		https://www.mozilla.org/firefox/76.0.1/whatsne	{"oldversion":"	{"visit_type":"temporary r	{historyvisit={"id":"4","places		E:\testcase\
url	5	05/14/2020	05/14/2020		https://www.mozilla.org/en-US/firefox/76.0.1/w	{"title":"Whatâ	{"visit_type":"permanent i	{historyvisit={"id":"5","places		E:\testcase\

The tool will try to show all the associated fields for each record. Those that are not normally looked at, are shown in the 'extra fields' column, where each field is annotated by a 'name of field/value of data' pair. Many of the items of interest such as timestamps and URL have their own dedicated columns.

If running the tool with either the **-carve** or the **-parse_chunk** options, the 'data record sources' field will be populated with the offset of the record. For example, running the same command above but specifying carve as the parse algorithm, yields the same data above, but with the *data record sources* field populated.

```
>msp64 -db c:\dump\places.sqlite -out results.csv -carve
```

data record source(s)
{moz_historyvisits=["src":"carve main"; "record offset":"0x0005ffee"];moz_places=["src":"carve main"; "record offset":"0x0001fccf"]}
{moz_historyvisits=["src":"carve main"; "record offset":"0x0005ffdb"];moz_places=["src":"carve main"; "record offset":"0x0001fbfa"]}
{moz_historyvisits=["src":"carve main"; "record offset":"0x0005ffc9"];moz_places=["src":"carve main"; "record offset":"0x0001fb72"]}
{moz_historyvisits=["src":"carve main"; "record offset":"0x0005ffb5"];moz_places=["src":"carve main"; "record offset":"0x0001faec"]}
{moz_historyvisits=["src":"carve main"; "record offset":"0x0005ffa1"];moz_places=["src":"carve main"; "record offset":"0x0001f9f2"]}

This gives one the location information necessary to analyze the data in a hex editor to verify the results, if desired.

3.2 Integrated Parsing Algorithms

The **msp** tool offers three possible parsing algorithms to choose from; these are outlined below:

1. *Default* option. This option uses the internal SQLite library that is statically linked into the tool to perform a *SQL-Select* statement on the database under analysis. It is sensitive to corrupt databases.
2. *Carve* option. (**-carve**). This option uses a TZWorks based set of algorithms to traverse the SQLite data structures to parse the records in the database. It relies on the database's schema and internal tree-based structures to find the data. This option appears to work fine even if the database cannot be opened via the standard SQLite library. When corruption is present, this option will skip bad records and will attempt to go to the next one. It also looks at unused space for any records that may be present using the **-incl_slack** option.
3. *Signature-base* option. (**-parse_chunk**). This option *does not* make use of the SQLite schema or tree-based structures in the database to locate records. Instead, it looks for pre-defined signatures in order to locate records and parse them. Empirical testing has shown this approach works from either a fully intact database, a corrupted database or a partial blob of a database. While this option can pull valid records, it truncates the data when a record spans multiple SQLite-pages. For any records that are truncated, the output will be annotated with a flag identifying it as such.

3.2.1 Algorithms and their Pros/Cons

The benefit of the *default* option is its usefulness for verification and validation purposes. Given that the tool can produce the same output for any of the three available parsing options, one can use the *default* option as the base option to compare other parsing algorithm results. In this way, one can easily verify whether the *carve* option and/or *signature-based* option works, simply by comparing the results to that of the *default SQL-Select* option.

In most cases, the *carve* option (**-carve**) is a better choice over the *default* option, simply because it returns the same, if not more, results. If invoking the sub-option **-incl_slack**, the tool has the ability to detect unused space and switches to a *signature-based* scan for those areas.

Surprisingly, the *signature-base* option (**-parse_chunk**) competes very well with the other two options. Keep in mind, this option relies strictly on unique signatures being accurate for its success. While the other two options can dynamically adjust their parsing engine based on the schema identified in the database, the *signature-based* option cannot. Depending on the number of recoverable records in the database, it is possible for signature-based option to extract more records than the other options, however, the user is cautioned, that more records do not necessarily mean accurate data. For example, if one passes in a file that contains the contents of a disk volume, with the intent of extracting all the Firefox artifacts from that image, then the user may get multiple false positives on certain table records. The *msp* tool does a good job of statistically pulling out table entries that have many fields versus those tables that only have a few fields. Therefore, certain table entries will have less false positives than others.

The other issue to consider with the *signature-base* option is the merging operation from data in one table to another table (based on some relationship between the tables) may or may not make sense.

For example, if a timestamp from one table is merged with data from another table, and the data is not in sync (from a chronological point of view), then the resulting merged record will mislead the investigator of an event's occurrence time-wise. The other pitfall with the *signature-based* scans, which was mentioned earlier, is that approach will truncate the data if a record overflows into multiple databases pages; the *signature-based* scan will only report on data found in the initial page.

To handle the data accuracy issue, refer to the section on “*Merging of Data between Tables*”. In conclusion, despite the negatives for the *signature-based* parse, it is the only choice if analyzing partial chunks of database fragments, whether from memory or disk images.

3.3 Modified CSV Output

When parsing various databases, where a database type can have differing tables and each table translates to differing schemas or fields, one of the challenges in report generation is how can one get all the varying data fields into a common CSV format. The simple answer is to invoke the *Log2Timeline* option (**-csvl2t**), or the *Sleuthkit BodyFile* option (**-bodyfile**). These are excellent options to achieve this, since these formats have custom pre-defined fields. They are defined in such a way, so that the format allows for dissimilar datasets by assuming all record will have at least a timestamp and description of the event that occurred. These formats also contain fields for generic data for notes and comments.

The above formats, because of their nature, can take one record and create multiple CSV entries if an entry contains multiple differing timestamps. Therefore, if one desires to output a single CSV line per record, then some of the fields need to be designated as variable in nature. Leveraging off of the concept of the **-csvl2t** format, one can accomplish this by creating some static fields as well as some general-purpose fields. For the default or the **-csv** option, the *msp* tool does just that. Specifically, there are a few static fields where the types are set, but there are others where a quasi-JSON format is used. In this way, many of the fields of a record can be outputted in a way where like-fields, such as *Type of record*, *RowID*, *Timestamp*, and *URL* are static, but the other general-purpose fields can contain differing types of data. For general-purpose data, the quasi-JSON format used by the *msp* tool consists of outputting the data in a *name/value* pairing relationship.

3.4 Type Designations

The output will render two types of designations. The first is as result of merging records from tables in the accordance with the schema of the database. For this case, the following designations are used:

Record Type	Table(s) where the data resides	Database where the table(s) reside
Download	moz_annos or moz_downloads	places.sqlite or downloads.sqlite
Favicon	moz_icons or moz_favicons	favicons.sqlite or places.sqlite
Keyword	moz_keywords	places.sqlite

Bookmark	moz_bookmarks	places.sqlite
Url	moz_places, moz_historyvisits	places.sqlite
Origin	moz_origins	places.sqlite
Cookie	moz_cookies	cookies.sqlite
Formhistory	moz_formhistory	formhistory.sqlite

Alternatively, if merging of records from tables is turned off (via **-no_table_merge**), then the type designations may specify be the actual table name where the data came from. These table names are shown in section on “*Databases targeted by this tool*”.

In addition to the record types shown above, there are some cases where the type is supplemented with an extra word, such as **Trunc**, which means the data was truncated. This only occurs with using the *signature-based* scan (**-parse_chunk**). This is because the data in the record spans multiple database pages and for *signature-based* scans, only the data in the initial page is parsed.

3.5 Processing Multiple Databases

If desiring to process many database files in one pass, one can put the artifact databases in separate subdirectories that share a common parent folder (or just enumerate them on a live system) and use the **-pipe** option like so:

```
>dir c:\dump\firefox_dbs /b /s | msp64 -pipe -out results.csv
```

To be more discriminating one can use the **-enumdir** option along with the sub options **-num_subdirs** and **-filter** like so. This allows one to target a certain level of subdirectories and only files with the extension *sqlite*.

```
>msp64 -enumdir c:\dump\firefox_dbs -num_subdirs 10 -filter "*.sqlite" -out results.csv
```

The above command will process all the databases contained in the *c:\dump\firefox_dbs* folder and subfolders. The results of parsing all the databases found will be put into the file *result.csv*. To help distinguish which lines corresponds to which database file, an extra field is appended to each record identifying the source database.

3.6 Merging of Data between Tables

Certain tables contain relationships between them, where data from one table is meant to be combined with another table in order to populate all the fields for a record. The relationships between the Firefox database tables are shown in the section on “*Databases Targeted by this Tool.*” The **m_{sp}** tool will, by default, try to use these relationships and merge the data between the tables appropriately. Each merged dataset will be treated as a separate record to be outputted into the report. For example, if the records from three tables make two records after the data is merged, only the two merged records will be outputted by this tool in the report.

On the flip side, if one has two tables to be merged and they have a ‘one to many’ or ‘many to one’ relationship, then the tool will try to create a ‘one-to-one’ relationship in the results that are outputted. A good example is the *places.sqlite* database, where a ‘one to many’ relationship exists with the *moz_places* table and the *moz_historyvisits* table. One *moz_places* record can have one or more *moz_historyvisits* records. Since the *moz_historyvisits* record has its own timestamp when the visit occurred, to create a proper timeline of events, one needs to duplicate the *moz_places* record data to account for all the *visits* record data. This action of duplication of data from the *moz_places* record creates the ‘one-to-one’ relationship in the output. Unfortunately, this gives the perception that there are a large number of duplicate records. Whether it be with the *moz_places* to *moz_historyvisits* relationship or some other table to table relationship, inevitability, there will be duplicates where some of the records outputted will match each other, especially when considering parsing deleted records out of unallocated space. This tool does not make the determination whether the records it parses are duplicated or not; it just outputs all the data.

In some cases, one may not want this merging to take place, and may want to see all the un-merged data from each table separately outputted as a separate record. This behavior can be done by invoking the **-no_table_merge** switch. This option only works with the *default* or **-csv** output modes (and does not work with **-csvl2t** or **-bodyfile**). This is because not all table records that are parsed by this tool have a timestamp associated with them, which the **-csvl2t** and **-bodyfile** formats rely on.

The main *use-case* for the **-no_table_merge**, is when one processes chunk of data (i.e. consider a partial memory dump, volume dump or a partial database file) which contains some Firefox artifacts. In this case, any records extracted from partial tables may relate to one computer’s account Firefox data, but not to another account. Alternatively, using the same example, assume there is only one user account on the computer; what could happen is that a parsed timestamp from one table may be out of sequence, from a chronological perspective, from data in another related table. Therefore, any merge operation in the above cases is dubious at best, since there is really no good way to tell if the merge operation will yield accurate results.

3.7 Parsing Firefox Artifacts from Memory or a Disk Image

If one wishes to parse artifacts from a file-based archive that contains a memory or a disk image, then one would use the **-parse_chunk** option. During the parsing operation, the tool uses a *signature-based* scan looking for records. Below is an example of performing this operation on a VMWare memory image. Notice we incorporated the **-no_table_merge** option as well, since we do not want to merge table data together. This is done as a precaution in case there were multiple instances of Firefox artifacts at one time or another; each instance, in this case, would represent a different user account on the system. Merging table data from one user to another user would yield incorrect and misleading results.

```
>misp64 -db c:\dump\test_image.bin -parse_chunk -no_table_merge -out results.csv
```

Notice in the command shown, that we still use the **-db <file>** syntax even though the file we are parsing is not a database, but is an image of physical memory stored as a file.

The same type of scan can be done on any image that is not encrypted. The only restriction here is that the image (memory, volume, disk or chunk of data) has to be identical to the system it came from. The key here is the SQLite records being scanned/parsed need to be preserved in their original form.

The last point to mention is if the **misp** tool detects a *very large* file is being processed for analysis, it will complain if you are not using the option **-parse_chunk**. Also, **misp** will complain if either the **-csvl2t** or **-bodyfile** output options are used for *large file* analysis, since only the **-csv** (or the default) output option is allowed for this situation. This limitation is hardcoded into the tool. Furthermore, it will automatically switch into the mode **-no_table_merge** for very large files. The term ‘*very large*’ in this context are sizes not normal for individual Firefox databases, so an arbitrary size above 130 MB is used for this threshold.

3.8 Bypassing the Embedded SQLite library

The **misp** tool has the SQLite library embedded into the binary. More information about this is discussed in the section *Use of the SQLite Library*. The **misp** tool makes use of this library in the default mode when parsing.

Sometimes, however, one may not wish to use the SQLite library for analyzing tables and extracting records, so an option was added to bypass the SQLite library and use the *TZWorks* internal SQLite algorithms to parse the database. This functionality can be invoked in one of two ways: (a) with the **-carve** option or (b) the **-parse_chunk** option. Out of the two options, one should opt for the first, the **-carve** option. This option will try to traverse the internal SQLite data structures in the database (even corrupted ones), and should extract all the same information as if using the normal SQLite. The

difference here is the **-carve** option is more immune to database corruption or database lockdown than the *default* option.

The purpose for the second option **-parse_chunk**, is to go a step further and operate on only a subset of the database. More specifically, if at least a page of the database is available, this option will try to make sense of any records it finds. The limitations of this option include: (a) it will not be able to handle overflow records between SQLite pages, and (b) it may not be able to provide joins between tables that have a relational aspect. The **-carve** option discussed earlier, however, will handle the overflow of data between pages and perform the necessary joins between tables that have dependencies between them. The benefit of the **-parse_chunk** option is that it can handle pulling out records from a journal file independently of the main database file, whereas the other two options cannot.

3.9 Splitting the Mozilla Sessions into Separate Files

One of the use-cases requested was to run a parsing tool against a system with multiple accounts and breakout the parsing results by account into separate files. Initially added with the companion **mcp** tool (for processing Mozilla Cache files), this capability was extended to this tool. The option is **-split_sessions** and it can be used with the directory enumeration options (**-enumdir** or **-pipe**). This option 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. This assumes that the starting folder includes the user's account folder/subfolders. Below is an example using this syntax.

```
>mcp64 -enumdir c:\users -num_subdirs 15 -filter "*.sqlite" -split_sessions -out results.csv
```

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 "results") will be the part of the name with an incremented number along with the folder name used by Mozilla for that session.

3.10 Verification and Validation

All tools need to be tested with some form of verification to ensure their results are accurate. Part of that testing is to validate the tool's functionality across different artifact versions. If the tool developer can automate this testing, then it allows the developer to test the tool across many datasets quickly. This in turn quickly identifies inconsistencies and problems so that a wide range of bugs can be diagnosed and fixed.

Normally, the developer tries to do as much of this testing before sending a tool out to clients. In the case of Firefox, however, since it has a history of changing the schemas across versions so that they are not backwards compatible, we decided to temporarily add an option for clients to run this type of verification on their own, if they so choose. To this end, the *msh* tool incorporates the *-verify* option to aid in this purpose.

The *-verify* option internally invokes all three parsing engines in sequence to parse the same database so it can compare the results of all three. Simplistically, if all the results match, then the confidence is very high the tool is working as designed. If the results do not match, it will be because a version of Firefox is being analyzed where the tool may work with one of parsing engines, but not the others. The first parsing engine most likely to have problems will be the *signature-based* parsing, since it is more sensitive to schema changes. In contrast, the default *SQL-Select* type parsing engine should be the most robust if there are schema changes, because it will key off of specific field names, which typically are more consistent across versioning. Either way, the purpose of the *-verify* option is to provide an internal test to alert a user if any issues are found.

The nice thing about the way this option was implemented, is not only does it check the internal parsers against themselves, but it also outputs critical diagnostic data that can be used by TZWorks to help improve the tool. To ensure no personal information is outputted, the *-verify* option sanitizes the results so that it does not contain private/confidential information from the raw artifact. The output primarily contains metadata from the SQLite internal structures. This causes the data generated to be cryptic and only useful for machine type learning/statistics. An additional sub-option was added (*-add comments*) to annotate some additional commentary to the results; this provides some extra information for the user if a test passed or failed and why.

```
>msh64 -db places.sqlite -verify -add_comments -out verify_results.txt
```

As mentioned earlier, the data produced is mostly cryptic since it contains statistical information about the database and records being parsed. This statistical information, if sent back to TZWorks, will help us improve our parsing engines for future releases.

Below is a screenshot of one of the entries in the results after running this test. For each database processed, there will be information about the various table schemas of interest. From this we can see if the schema has been updated from one version to another. In addition, the output shows the number of records parsed by each engine, the signatures found, and so on.

```
// ----- file: places.sqlite -----
// CREATE TABLE moz_origins ( id INTEGER PRIMARY KEY, prefix TEXT NOT NULL, host TEXT NOT NULL, frequency INTEGER NOT NULL)
// CREATE TABLE moz_places ( id INTEGER PRIMARY KEY, url LONGVARCHAR, title LONGVARCHAR, rev_host LONGVARCHAR, visit_count INTEGER)
// CREATE TABLE moz_historyvisits ( id INTEGER PRIMARY KEY, from_visit INTEGER, place_id INTEGER, visit_date INTEGER, visit_time INTEGER)
// CREATE TABLE moz_inpuhistory ( place_id INTEGER NOT NULL, input LONGVARCHAR NOT NULL, use_count INTEGER, PRIMARY KEY (place_id, input))
// CREATE TABLE moz_bookmarks ( id INTEGER PRIMARY KEY, type INTEGER, fk INTEGER DEFAULT NULL, parent INTEGER, position INTEGER)
// CREATE TABLE moz_bookmarks_deleted ( guid TEXT PRIMARY KEY, dateRemoved INTEGER NOT NULL DEFAULT 0)
// CREATE TABLE moz_keywords ( id INTEGER PRIMARY KEY AUTOINCREMENT, keyword TEXT UNIQUE, place_id INTEGER, post_data TEXT)
// CREATE TABLE moz_anno_attributes ( id INTEGER PRIMARY KEY, name VARCHAR(32) UNIQUE NOT NULL)
// CREATE TABLE moz_annos ( id INTEGER PRIMARY KEY, place_id INTEGER NOT NULL, anno_attribute_id INTEGER, content LONGTEXT)
// CREATE TABLE moz_items_annos ( id INTEGER PRIMARY KEY, item_id INTEGER NOT NULL, anno_attribute_id INTEGER, content LONGTEXT)
// moz_places
// : Phase 1: Passed: carve option ID'd the same records that SQL Select option did.
// : Phase 2: Passed: Ordering of fields consistent.
// : Phase 3: Passed: Quick look comparison of values between each record of parsers found no mismatches
+ [0 : 0]
> [335-335-335] : [335-335-335]
* [335-335-335] : [017ffff00001e090314111308110415080708131805110b000614050e141103040f110e11] : [017ffff00001e090314111308110415080708131805110b000614050e141103040f110e11]
// moz_historyvisits
// : Phase 1: Passed: carve option ID'd the same records that SQL Select option did.
// : Phase 2: Passed: Ordering of fields consistent.
// : Phase 3: Passed: Quick look comparison of values between each record of parsers found no mismatches
+ [0 : 0]
> [387-387-387] : [387-387-387]
* [387-387-387] : [023f0000000c080305110f0b150815081204] : [7-774] : [10e212a16f119b18; 10a189f110e212a16f119b18; 10a189f110e212a16f119b18]
// moz_annos
// : Phase 1: Passed: carve option ID'd the same records that SQL Select option did.
// : Phase 2: Passed: Ordering of fields consistent.
// : Phase 3: Passed: Quick look comparison of values between each record of parsers found no mismatches
+ [0 : 0]
> [8-8-8] : [8-8-8]
* [8-8-8] : [04f70300001208030f0b0000d020e050b0417131803000b00] : [10-14; 11-2] : [10f21b1de18a2110f21b1de18a2110f21b1de18a211]
```

One final comment on the **-verify** option. This is not a do-everything type built-in test. While it is very capable and provides a wealth of information, the biggest limitation of this test is that it only compares un-merged tables records. Therefore, if there is an error during a merge operation between some *table-to-table* relationship, it is not included in the battery of tests used by the **-verify** option. The other testing shortfall is the last (phase 3) test only compare the first two parsing engines resulting values and doesn't consider the third parsing engine (signature-type scan). These shortfalls may be something added in the future, but for now the purpose of this automated testing is to: (a) capture differences in various Firefox formats, (b) identify issues with the various parsing engines in the tool so they can be fixed quickly, and (c) get more empirical results as it pertains to *signature-type* scanning, since this engine at its core relies on statistical data.

4 Use of the SQLite Library

The databases that are targeted by the **misp** tool are SQLite databases. For the purposes of the **misp** tool we statically link in the SQLite library to ensure the tool has minimal dependencies. The source code for the SQLite library is an amalgamation of the SQLite 'C' source files, version 3.32.3. More information about SQLite, the documentation and the source code can be seen at the official SQLite website [<http://www.sqlite.org/>].

Normally when we build a tool to parse a raw artifact, we prefer not to use outside libraries, however, in this case, the SQLite library has an option to open a SQLite database in 'read-only' mode. From the testing done and from the documentation, it appears that this is acceptable for this release.

5 CSV Field Names / Meaning

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

CSV Field	Definition
field	Cache version number
type	Type of data based on the table the record comes from. Example of types include: url, cookie, bookmark, favicon, download, etc
rowid	Internal parameter to the SQLite table record identifier
create time [UTC]	Date/Time the URL or item was created
last access [UTC]	Date/Time the URL or item was last visited or accessed
expires [UTC]	Date/Time the URL or item expires
url or name	URL or name of the item
params	Any HTTP parameters passed in with the URL (or can be used for other items if not a URL)
params translation	Translation of any parameter passed in (or can be used for other items if not some that require translation)
extra fields	Any fields not covered by the previous fields that are part of the record
data record source(s)	The source table and record offset within the database where this record was parsed (only applies to <i>-carve</i> and <i>-parse_chunk</i> parsing options)
file	Database file that was parsed

6 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 Firefox this tool has been tested on is v3.0.1. Therefore, prior versions should not work.
- The *-split_session* folder enumeration 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.

6.1 Versions of Firefox tested with *mvp*

As the version of the browser changes, inevitably so do the tables and their fields change. While not strictly an issue, in that the *mvp* tool will still try to parse the data as the schema in the changes, any new field(s) added in future versions of the database will not be reported on until the *mvp* tool is updated for newer schema changes. Below are the versions of the database tested.

Firefox Versions	Tested these tables (if applicable) with each Firefox Version
3.0.1, 4.0, 5.0, 11.0, 24.0, 31.0, 45.0, 47.0, 52.0, 60.0, 69.0, 72.0, 76.0, 77.0, 78.0, 79.0, 80.0, 95.0, 100.0, 102.0, 105.0, 119.0, 120.0, 121.0, 123.0, 124.0	moz_places, moz_historyvisits, moz_origins, moz_bookmarks, moz_inputhistory, moz_keywords, moz_annos, moz_items_anno, moz_anno_attributes, moz_cookies, moz_downloads (for older versions), moz_formhistory, moz_favicons (for older versions), [moz_icons, moz_icons_to_pages, moz_pages_w_icons] (for new versions).

7 Available Options

Option	Description
-db	Specifies which database file to act on. The format is: -db <database or file to parse>
-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.
-csv/2t	Outputs the data fields in accordance with the log2timeline format.
-bodyfile	Outputs the data fields in accordance with the 'body-file' version3 specified in the <i>SleuthKit</i> . The date/timestamp outputted to the body-file is in terms of UTC. So if using the body-file in conjunction with the mactime.pl utility, one needs to set the environment variable TZ=UTC.
-username	Option is used to populate the output records with a specified username. This only applies to the -csv/2t option. The format is: -username <name to use> .
-hostname	Option is used to populate the output records with a specified hostname. This only applies to the -csv/2t option. The format is: -hostname <name to 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 <code><"*.ext *partialname* ..."></code> . The wildcard character '*' is restricted to either before the name or after the name.
-no_whitespace	Output the date using the specified format. Default behavior is -dateformat <code>"mm/dd/yyyy"</code> . This allows more flexibility for a desired format. For example, one can use this to show year first, via <code>"yyyy/mm/dd"</code> or day first, via <code>"dd/mm/yyyy"</code> , or only show 2 digit years, via the <code>"mm/dd/yy"</code> . The restriction with this option is the forward slash (/) symbol needs to separate month, day and year and the month is in digit (1-12) form versus abbreviated name form.
-csv_separator	Only applies to -csv and -csv2t options. Used in conjunction with the -csv option to change the CSV separator from the default comma to something else. Syntax is -csv_separator <code>" "</code> to change the CSV separator to the pipe character. To use the tab as a separator, one can use the -csv_separator <code>"tab"</code> OR -csv_separator <code>"\t"</code> options.
-dateformat	Output the date using the specified format. Default behavior is -dateformat <code>"yyyy-mm-dd"</code> . 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 <code>"hh:mm:ss.xxx"</code> . One can adjust the format to microseconds, via <code>"hh:mm:ss.xxxxxx"</code> or nanoseconds, via <code>"hh:mm:ss.xxxxxxxxxx"</code> , or no fractional seconds, via <code>"hh:mm:ss"</code> . 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.
-carve	Experimental option. Bypass the SQLite embedded library and parse using TZWorks internal algorithms. This is useful when the database to be parsed is corrupted and the SQLite library has trouble parsing it.
-incl_slack	Experimental option to look at unused space to see if any records are present. Not required with the -parse_chunk option. Use this in conjunction with -carve or default option to look for discarded records.
-parse_chunk	Experimental option. Given a portion (chunk) of the database, this option will examine the data to see if any records exist and parse out the contents. This is a signature-based parse so it can parse out records from chunks of memory or slack space (in the form of a file).
-no_table_merge	This option is for pulling records from an image. It is also used for testing and debugging purposes. If you want to see all the tables that were parsed without merging any relationships, use this option.

-verify	This option is for testing and debugging purposes only. This option runs all 3 parsing engines in the tool (<i>SQL Select</i> parse, <i>Carve</i> parse and <i>Signature-based</i> parse) and reports whether the parsers work at least up to the level of the SQL Select parse. Metadata is generated that can be used to help develop more robust parsing algorithms.
-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 CSV output using this option.

8 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.

9 References

1. Mozilla-central Places Databases [<https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Places/Database>]
2. Mozilla Desktop Data Stores [<https://github.com/mozilla/firefox-data-store-docs>]
3. SQLite library statically linked into tool [Amalgamation of many separate C source files from SQLite version 3.32.3].
4. SQLite documentation [<http://www.sqlite.org>].
5. DB Browser for SQLite [<http://sqlitebrowser.org/>]