2018 11th International Conference on IT Security Incident Management & IT Forensics

Linux Memory Forensics: Expanding Rekall for Userland Investigation

Johannes Stadlinger* Frank Block*† Andreas Dewald*‡
*Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany johannes.stadlinger@fau.de
†ERNW GmbH, Heidelberg, Germany fblock@ernw.de
‡ERNW Research GmbH, Heidelberg, Germany adewald@ernw.de

Abstract

The field of memory forensics is getting more important in forensic investigations for obtaining valuable data of a running system. Besides kernel artifacts, there might be also plenty of interesting data in the heap of a user space process, but unfortunately, that area has not yet received the attention it deserves in the forensic field. This paper shows that the heap of user applications may also be a rich source of information including data like credentials that can be helpful in a forensic investigation. With the help of the *HeapAnalysis* plugins, previously published by Block, we examined the heap of selected Linux userland software and managed to identify data of interest and also certain application-internal structures, which link those data snippets together. The result of our analysis is a set of plugins for the Rekall framework, enabling an investigator to automatically extract process-related information such as login credentials, command history and file information for those applications.

Keywords

Digital Forensics; Memory Forensics; Userland; Heap; Glibc; Linux; Rekall

I. Introduction

Memory Forensics has grown rapidly in the last years [1], [2] and allows examiners to investigate volatile memory. Ligh et al. describe it as "*the most fruitful, interesting, and provocative realm of digital forensics*" [3]. Due to the increase of devices with full disk encryption, it gets more and more difficult for investigators to access and recover data of interest from a switched-off computer. That is one of the reasons why memory forensics is getting more important in forensic investigations for obtaining data of a *running system*. Besides forensically interesting artifacts like running processes, network connections, and executed commands [3], which are more or less easily extractable by well known tools like Rekall [4] or Volatility [5], there might also be plenty of interesting data hidden in the *heap* of a user space process (i.e. credentials, IPs, a command history, etc.). Existing plugins that extract data from user space processes typically treat the heap as one big bulk of data and try to identify data of interest by searching for a specific pattern. For example, Rekall's *bash* plugin searches the entire heap for occurrences of a #-character followed by a string containing a UNIX-timestamp (e.g., `#1510398671`) [3] in order to reconstruct the command history. However, that approach fails if no such patterns are available in the content of interest.

A. Motivation

The extraction of data from the heap has not yet received the attention it deserves from a forensic point of view. In the case of Windows applications, there is only a limited number of plugins available, which cover applications such as the Windows command line and Notepad (see [6], [7], [3]), while the number of applications in the context of Linux is even smaller (e.g. the bash plugin [8]). Due to the recent advances made by Michael Cohen for Windows [6] and by Block and Dewald for Linux [9], investigators now have a tool at hand to analyze the heap in a structured way and to build plugins that extract information of interest based on their analysis results (which has already been done exemplarily for the zsh shell and the Keepassx password manager [9]). So far, these new possibilities have not been used excessively and thus there is still a lack of tools that extract useful information from specific user space processes. Since forensic examiners need to rely on new tools [10], our goal is to adopt the approach of Block and Dewald [9] and to extent the set of available tools by analyzing the heap of a number of Linux userland applications and implement plugins for each application for the memory forensics framework Rekall, which is made publicly available along with this paper[1].

B. Related Work

The approach of searching the heap for known patterns in order to extract useful information is already adopted by existing plugins such as *cmdscan* [7] and *bash* [8]. The general problem with that approach is that it only works if the data of interest is marked by a searchable pattern. Another example is given by the work of Valersi [11], who focused on the Firefox browser and its private browsing sessions. By examining the process's memory, he was able to rebuild a data

[1]https://faui1-files.cs.fau.de/public/rekall-plugins/plugins.tar.xz

978-1-5386-6632-6/18/$31.00 ©2018 IEEE
DOI 10.1109/IMF.2018.00010

structure to extract forensically relevant information regarding those sessions. As a result, he built a proof of concept plugin for Rekall that prints all visited URLs of a private browsing session, which however also depends on specific patterns.

In 2014, Ligh et al. [3] published a more advanced approach. This was the first time that investigators used deeper knowledge about the inner structure of the heap implementation in order to retrieve information from user space applications. They wrote a plugin for Volatility that is able to isolate single heap chunks of *Notepad*'s process memory in order to extract text fragments the user has written into an open document.

The work of Ligh et al. [3] is very specific to *one* concrete user space application, but as forensic examiners need a more general tool for the analysis of applications, Cohen in 2015 [6] presented several plugins for the memory forensic framework Rekall that allow an investigator to analyze the heap in more detail. It changed the view on the heap of Windows applications from a large bulk of data to single chunks (see Section II-A), which are more easy to analyze.

Since the work of Cohen focuses just on Windows and its heap implementation, Block and Dewald [9] gave a new perspective for investigating Linux userland software in 2017. They analyzed the heap implementation of the GNU C Library (glibc) [12] and similar to Cohen, built a set of plugins for the Rekall framework. These plugins are the foundation for this work and are described in more detail in Section II.

C. Contributions

With this work, we want to show that – besides already extractable kernel related information – the heap of user applications may also be a rich source of information including data like *credentials* that might assist the investigators in a forensic investigation. By using the already developed plugins by Block [9] (i.e., *heapdump* and *heapsearch*), we analyzed the heap of some Linux userland applications that rely on the *glibc* heap implementation. Those applications have been chosen because they reveil additional information that might give the examiners access to further traces and evidence. Finally, we implemented dedicated plugins for each of them. The following list shows the plugins and the data they extract:

- **curl**: URL, Name of the output-file, Username and Password.
- **gnome-keyring-d**: Meta-data about existing keyrings, Name of the stored password, SSH-private keys (ASCII-armored).
- **seahorse**: Name of the stored password, PGP and information, SSH and information.
- **ssh**: User- and Hostname, Src. and Dest. IPv4 address.
- **sshfs**: Partial filelist, User- and Hostname, name of the server and local folder
- **pwsafe**: Group names, Name of the password entries, username, password (if displayed).
- **sqlite**: Command history, table schemes.
- **owncloud**: Username and password, hostname, synchronisation protocol.

D. Outline

This work is structured as follows: In Section II, we provide an introduction to the *HeapAnalysis* components developed by Block [9] and how they are used within our work. Section III provides an overview of the recovered datastructures and the corresponding plugins that extract that information from the process heap. Section IV evaluates our work and the final Section V concludes our work and provides limitations as well as future work.

II. Fundamentals

This section describes the *HeapAnalysis* class, functions and plugins used in the context of this work.

A. Chunks

Chunks are little (or also large) portions of memory that typically contain data such as char arrays, structs and instantiated class objects. They are described by the *malloc_chunk* struct, which at least contains the chunk's size. There are several types of chunks, but for the context of this work, we mainly differentiate between *allocated* and *freed* chunks. *Allocated* chunks are still in use and *freed* chunks are marked as free, which means they can be overwritten with new content.

B. The HeapAnalysis Class

All classes and plugins covered in this and the following section are part of the current stable version of Rekall[2] (version 1.7.1). The main component for all plugins is the *HeapAnalysis* class, written in Python. It is responsible for reconstructing the heap of a given application by searching for specific meta information and essentially provides several methods for the access to all chunks.

To start the analysis, the first function to use is `init_for_task`, which initiates the heap reconstruction process. As soon as this function finishes, the initialization is complete and the chunks can be accessed with several functions. For example,

[2]https://github.com/google/rekall

`get_all_chunks` returns all chunks that can be found in the memory, including all *allocated* and also *freed* chunks. To only get chunks of a specific type, functions such as `get_all_allocated_chunks` and `get_all_freed_chunks` can be used.

To ensure the proper functionality of the *HeapAnalysis* class, some debug information for the glibc version in use are necessary. While the *HeapAnalysis* class is equiped with such information for several versions and also some autodetection algorithms, there might be instances in which the information is not accurate or the autodetection routines fail. In such cases, the investigator can and should provide this information for the correct glibc version for his case. These debug information include struct details, such as the *malloc_chunk* struct and also offsets to two struct instances located in the loaded glibc library: *main_arena* and *malloc_par*. For more details, see the Technical Report [9] of Block et al., or the actual source code[3], respectively.

C. Heap Analysis Plugins

Based on the *HeapAnalysis* class, Block developed several Rekall plugins for the analysis of an application's heap: *heapinfo*, *heapdump*, *heapsearch* and *heaprefs*. They support the investigator in performing a manual analysis by providing an easy to use interface, which essentially searches through, and prints content from chunks of interest. With the help of these plugins, we are able to find chunks with specific content, identify connections between chunks and dump their content for further analysis. The following list gives a rough overview of each plugin used in the context of this work.

- *heapinfo*: This plugin prints a short summary about the number and size of all chunks and some meta information, which serves as a first overview of the target process.
- *heapdump*: This plugin gives the examiner the possibility to dump all chunks of a target process into a specific directory on a local machine. Each chunk is stored as a single file with a unique name pattern including the *PID*, type of chunk, the *offset* of the according *malloc_chunk* struct and information about its size.
- *heapsearch*: The *heapsearch* plugin helps in searching for specific chunks and can be divided in two basic functionalities. The first one allows to search all chunks for a specific *string*, *regex*, or *pointer*. The second expects a concrete address(es) of an already identified chunk, provided via the `--chunk_addresses` parameter. All chunks that contain a reference to the given chunk are printed. This case is illustrated in Figure 1, where *Chunk #1* and *Chunk #2* contain a pointer to the *Target Chunk*.

Figure 1: Functionality of a `heapsearch` call. As a result, all chunks pointing to the `target chunk` are printed.

- *heaprefs*: The *heaprefs* plugin examines the content of a chunk and searches for pointers to other chunk(s). Figure 2 gives a graphical representation of its functionality. The target chunk contains a pointer to chunk #1 and #2.

Figure 2: Functionality of a `heaprefs` call to get all chunks, the `target chunks` refers to.

These plugins can be used to to analyze the heap of some applications and to create specific plugins, as described in the following section.

III. Analysis and Plugins

In this section, we describe the results of our analysis of each of the selected user land applications and our implemented plugins that use those results to extract valuable information from memory dumps.

A. cUrl

We examined two different versions depending on the architecture:

- 32-bit: version: 7.52.1
- 64-bit: version: 7.53.1

We created several test scenarios where we downloaded files and protected some of them with a *username* and *password*. In the following sections, we report on the results of our manual analysis of the memory of *cUrl*'s processes.

[3]https://github.com/google/rekall/blob/master/rekall-core/rekall/plugins/linux/heap_analysis.py

1) URL and Output File: We start the analysis by focusing on actual information like *which* file was downloaded from *where* and, if it was written somewhere else than stdout, *what* is the provided filename.

By applying the *heapsearch* plugin, we gathered *five* different chunks with the same size whose only content are the entire URL string. In our case, those containers have a size of 96 bytes, but after further analysis with shorter URLs, we noted that they have at least a size of 32 (x32: 16) bytes. After searching for references to those *five* chunks, three of them could be eliminated from the sample space, because one is not references, and two others are referenced by fragments with a size greater than 4000 bytes which makes them unreachable for further examination. So there are two remaining chunks in the sample space: The first one has a size of 32 bytes and points to the string at offset 0. The second one has a size of 48 bytes and holds at the one hand a pointer to the string at offset 8 (x32: 4) and a second pointer at offset 16 (x32: 8).

We examine the remaining pointer at offset 16 (x32: 8). As one can see in the hexdump listed in Listing 1, it contains the name of the output file the user passed to cUrl (option `-o <name>`). The size of the container may differ since it depends on the length of the included name. The string starts at byte 0 and is again terminated by null bytes.

Listing 1: Hexdump of the chunk containing the output file.

```
00000000  6f 75 74 70 75 74 64 75  6d 6d 79 2e 66 69 6c 65  |outputdummy.file|
00000010  00 fb c4 01 00 00 00 00                           |........|
```

2) Credentials: In case, credentials have been provided by the user as arguments (option `-u user:password`), it might be valuable to recover them. If both are passed during calling cUrl it is easy to extract them by examining the console's history. But it gets more complicated if the user is asked for the password during runtime which we focused on and cover in the following.

During our previous examination, we detected a container that refers to the URL and the filename. After applying *heapsearch* it reveals a single chunk of a size of 1104 (x32: 616) bytes. That size stays constant over our analysis. It contains *six* pointers in total, whereas a bunch of three consecutive refer to the already examined container (offset 432, 440, 456, in x32: 252), a single pointer at offset 288 (x32: 172) refers to a chunk of a size of 32 bytes. By dumping the content of that chunk, we recognized a string that includes the *username*, followed by a colon and the *password*. To verify that result, we ran the analysis with different scenarios.

The final developed structure is shown in Figure 3. The chunk that refers to the URL and filename has a pointer at offset 432. The credentials consisting of username and password are in the container covered by the pointer at offset 288.

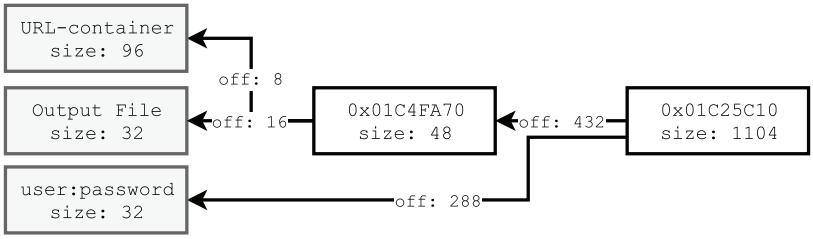

Figure 3: Illustration of the developed structure to retrieve *username*, *password*, *outputname*, *url*.

3) Plugin 1: `curl`: The resulting plugin that recovers all previously mentioned data structures is called *curl*. A sample output is given in Listing 2. It provides a short set of information to the forensic examiners. In the first column, the current PID of the process is given, followed by the url address the user has typed in. Additionally, if the user has entered a optional location where the file is to be stored, it is also printed on the examiner's console. Finally, the plugin is able to recover the sensible information like *username* and *password* which are printed in the two remaining columns.

Listing 2: Resulting output of the *curl* plugin. All previously gathered information are printed in a table.

```
 pid                    url                                          output
------ ---------------------------------------------- --------------------------------
 1068  https://pool.c0nf.de/curl/2Gb.dat              outputdummy.file

          user                    password
---------------------- ------------------------
mem_user                mem_password
```

B. gnome-keyring-d

The first password manager we want to focus is the *gnome-keyring-d*. We created a test scenario consisting of several keyrings where each holds a different amount of passwords (closed or open). Furthermore, we added SSH keys. The examined versions of the applications are:

- 32-bit: version: 3.20.0
- 64-bit: version: 3.20.0

As in all password managers, the most wanted artifact to recover is the *master password* of an existing keyring or container. With the help of that, the investigator would be able to decrypt all further underlying password entries and so they would have complete access to the suspect's login credentials. All attempts recovering such information failed during the analysis. Furthermore, we were not able to recover any information regarding actual password phrases of entries. So, we have to concentrate on keyrings and what is recoverable about their contents. Furthermore, we take a closer look on SSH keys that are stored within the gnome-keyring-d.

1) Keyrings: Names and number of members: We started our analysis by searching for actual names of keyrings which we expected to be in the memory. Therefore we invoked *heapsearch* and got a different amount of potential chunks back for each ring depending on the underlying test file. The identified strings consists of one of the following two patterns:

- `/org/freedesktop/secrets/collection/`**`<ringname>`**
- `/org/freedesktop/secrets/collection/`**`<ringname>`**`/`**`<number>`**

The occurrence of the first pattern differs from entry to entry, but what we could verify is that it appears at least *once* for each ring. After running several test scenarios, we could also assume, that it has at least a size of 80 bytes and the string is encoded as *utf-8*. Furthermore, it starts at byte 0 and is terminated by null bytes. Additionally, we could say, that *all* name of the rings are present, despite the fact whether they are *closed* or *open*. As a connection, we recognized that one single chunk refers to all keyring names.

The second pattern represents the number resp. the ID of a keyring's entry. This could be verified after running several scenarios and observing the number of appeared potential chunks. Additionally, we observed, that for each entry of a keyring, there exists at least *one* chunk with such a pattern. With the help of *heapsearch*, we were able to retrieve *one* chunk that refers to all occurrences of that pattern. We have to admit, that these pointers are not aligned in a obvious pattern. But with the help of that combining component, we are able to describe *how many entries a keyring consists of*.

Finally, we searched for a clustering component that combines the two above mentioned insights. After applying *heapsearch*, we discovered one common container. It has a size of 1024 bytes (x32: 512). It consists of several pointers that are clustered in a group of three. All cluster are 96 bytes (x32: 56) apart. Figure 4 concludes the here developed structure.

2) Stored notes – Entries of the keyrings: The following approach is only possible, if the keyring, in which the entry is stored, is open at the time of acquisition. Otherwise all information is freed and thus not reachable.

The search for actual password entries were without any reliable results. The only mentionable case would be if the user has previously displayed the requested password. But it is still observable for a short amount of time and is freed afterwards.

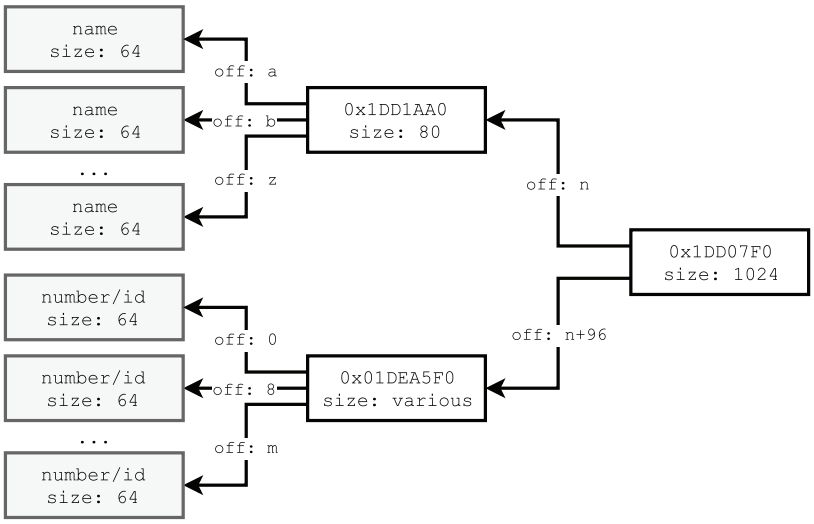

Figure 4: Illustration of the developed structure to retrieve the information regarding meta infos about the existing keyrings.

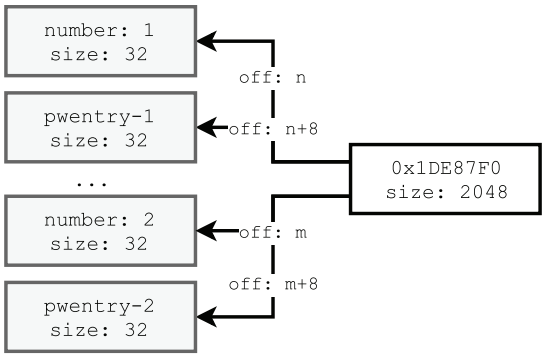

Figure 5: Illustration of the developed structure to retrieve the information regarding the entries of an open keyring.

Listing 3: Hexdump of the chunk containing the *private* key of a SSH key.

```
00000000  2d 2d 2d 2d 2d 42 45 47  49 4e 20 52 53 41 20 50  |-----BEGIN RSA P|
00000010  52 49 56 41 54 45 20 4b  45 59 2d 2d 2d 2d 2d 0a  |RIVATE KEY-----.|
00000020  4d 49 49 4a 4b 51 49 42  41 41 4b 43 41 67 45 41  |MIIJKQIBAAKCAgEA|
[...]
00000c80  66 43 75 74 69 61 74 2b  39 79 47 76 0a 2d 2d 2d  |fCutiat+9yGv.---|
00000c90  2d 2d 45 4e 44 20 52 53  41 20 50 52 49 56 41 54  |--END RSA PRIVAT|
00000ca0  45 20 4b 45 59 2d 2d 2d  2d 2d 0a 00 00 00 00 00  |E KEY-----......|
00000cb0  00 00 00 00 00 00 00 00                           |........|
```

So we focused on the stored *names* resp. *notes* of the entries because they often represent the name of the account or possibly even the username.

The *heapsearch* plugin results in exactly *one* chunk per entry. After running several scenarios we could derive that it has at least a size of 32 bytes (x32: 16) and contains one string that represents the name of a stored entry. As a container that refers to all or at least one entry, we identified – with the help of *heapsearch* – one chunk with a fixed size of 2048 bytes (x32: 1024). If there are more entries, more such chunks are allocated. After the examination with *heaprefs*, we recognized several pointers which are aligned in a specific pattern. Additionally, we were able to identify clusters of pointers for each stored note. The first one refers to a container with a single integer that acts as the internal ID for that entry. The second pointer leads to the actual content of the note. By combining those two insights, we are able to retrieve each entry of all keyrings.

Figure 5 concludes the here developed insights. There are several chunks with a size of 2048 bytes (x32: 1024) that hold pointers to each entry and its corresponding number.

3) SSH keys: For each stored SSH key pair, there exists a single chunk, that contains the entire ASCII-armored *private key*. To verify that assumption we ran several testing scenarios. After passing several substring of the key to *heapsearch* we got – as expected – exactly one chunk back with a size >2000 bytes. We could verify, that the entire private key is present (see shortened hexdump in Listing 3). Its size may differ, depending on the used algorithm and keysize. We recognized during our investigation, that there are chunks that hold the *name* of the corresponding private key, but no creation of linkage to the actual key was possible. So, to recover at least the key, we use a naive string-search approach by searching all chunks that start with the actual `-----BEGIN RSA` statement.

4) Plugin 2: `gnome_keyring`: The resulting plugin for the Rekall Framework is called *gnome_keyring*. It adopts all here mentioned insights about the process's inner structures. The output of the plugin is given in Listing 4 which is printed to the examiner's console after executing. It could be separated into three different sections. The first one recovers meta information of existing keyrings. This includes data like the name of the ring and its total amount of stored entries. It is irrelevant if the ring if opened or closed. That information is always retrievable. The second section summarizes all information regarding open keyrings. That includes the actual *name* and its id within the ring for each entry. Finally, the plugin supports the investigator by revealing the stored private SSH-keys of the user which are in the third section.

C. Seahorse

Seahorse is a password manager and provides a graphical user interface for the gnome-keyring. We examined the following versions depending on the architecture:

Listing 4: Resulting output of the *gnome-keyring-d* plugin.

```
  pid    entry               name                    type                value
  ------ ----- --------------------------- ----------------- -------------------------
Recovered name of keyrings with the numbers of entries it contains:
   989     1   nebenring                   Keyring           Entries in total: 3
   989     2   newring                     Keyring           Entries in total: 20
   989     3   hauptring                   Keyring           Entries in total: 6

Recovered name of keyring entries:
   989     1   entryentryentryentry-1      Stored Note       Number in keyring: 1
[...]

Recovered Private SSH keys (ASCII armored):
[...]
```

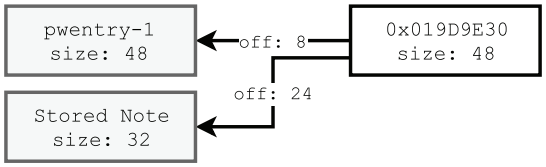

Figure 6: Illustration of the developed structure to retrieve the information regarding a Note/password entry.

- 32-bit: version: 3.20.0
- 64-bit: version: 3.20.0

At first, we tried to find information (e.g. *name of the keyring* and *name of actual entries*) about *closed* keyrings. All attempts, searching for such entries led to no results. So, a keyring that is closed could be seen as out of scope for further analysis. The artifact with the *highest* priority to recover is the *master* password of an actual keyring. After examining the chunks with *heapsearch*, no potential fragments are discovered.

1) Password entries: In that section, we want to take a step back and search for *entries* stored in an actual opened keyring. Unfortunately, no actual passphrases are present within the heap of that process. But what we could observe during our analysis, is that if Seahorse displays the user an actual password, the string is present for a short amount of time in one chunk. But that container is freed quickly after the user closes the window were it is shown.

At least, we are able to recover the actual *name* of each entry which could be used to provide a describing overview of the keyring. A single chunk with at least a size of 32 bytes (x32: 16) could be identified by *heapsearch*. It contains the name as a string starting at offset 0 and which is terminated by null bytes (see Listing 5).

Listing 5: Hexdump of the chunk containing the *name* of a password entry.

```
00000000  70 77 65 6e 74 72 79 2d  31 00 7b 01 00 00 00 00  |pwentry-1.{.....|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  30 00 00 00 00 00 00 00                           |0.......|
```

As a next step, we tried to find some sort of clustering or at least a list that could be traversed in order to discover all entries of a specific keyring. After invoking *heapsearch* we received exactly *one* chunk with a fixed size of 48 (x32: 24) bytes for each entry. It holds *five* pointers in total where the one at offset 8 (x32: 4) leads to the string. Additionally, at offset 24 (x32: 12) it references always to a string whose content is `Stored Note`. Figure 6 concludes the here developed structure for retrieving one entry of a keyring.

2) PGP keys: For each stored PGP key, we were able to identify exactly one chunk that contains the *mail address*, *name*, and if provided by the user a *note* (see Listing 6). The string starts after 88 bytes (x32: 44) and its size depends on the actual content and length of the string. The chunk is referenced by a container with a fixed size of 112 bytes (x32: 56). The output of *heaprefs* is given in Figure 7. As one could see, it contains several pointers, whereas the ones at offset 56 and 72 (x32: 32, 40) leads to the already identified string.

The remaining pointers are also of interest. The one at offset 48 (x32: 28) leads to a chunk with a fixed size of 112 bytes (x32: 56) which holds another reference to a chunk (offset 56, x32: 40) that contains the fingerprint of the corresponding public key (see Listing 7). The string starts at offset 0 and has a length of 40 bytes.

The same structure could be observed after analysing the remaining last pointer at offset 64 (x32: 36). But instead of the

Listing 6: Hexdump of the chunk containing the *name*, *mail*, and *notes* of a pgp entry.

```
...
00000040  f0 96 7b 01 00 00 00 00  00 00 00 00 00 00 00 00  |..{.............|
00000050  00 00 00 00 00 00 00 00  48 61 6e 73 20 57 75 72  |........Hans Wur|
00000060  73 74 20 28 74 65 73 74  29 20 3c 68 61 6e 73 2e  |st (test) <hans.|
00000070  77 75 72 73 74 40 65 78  61 6d 70 6c 65 2e 63 6f  |wurst@example.co|
00000080  6d 3e 00 48 61 6e 73 20  57 75 72 73 74 00 74 65  |m>.Hans Wurst.te|
00000090  73 74 00 68 61 6e 73 2e  77 75 72 73 74 40 65 78  |st.hans.wurst@ex|
000000a0  61 6d 70 6c 65 2e 63 6f  6d 00 00 00 00 00 00 00  |ample.com.......|
000000b0  c0 00 00 00 00 00 00 00                           |........|
```

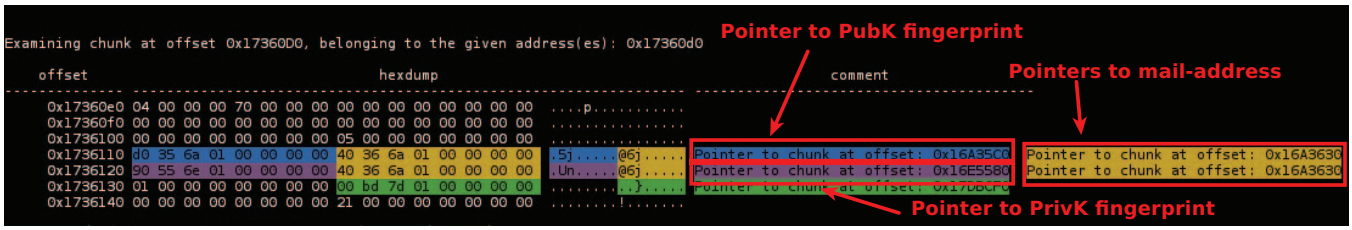

Figure 7: Analyzing a chunk's references by calling *heaprefs*.

public key's fingerprint, we were able to retrieve the one for the private key. Figure 8 illustrates our here presented results. Unfortunately, we were not able to discover any fragments of the passphrase.

Listing 7: Hexdump of a chunk containing the *fingerprint* of an encryption key (PGP).

```
00000000  36 36 44 44 33 35 36 36  31 46 45 31 36 39 35 42  |66DD35661FE1695B|
00000010  39 32 46 35 42 42 46 44  32 44 42 31 38 35 31 38  |92F5BBFD2DB18518|
00000020  41 31 41 31 46 36 31 46  00 00 00 00 00 00 00 00  |A1A1F61F........|
00000030  a0 00 00 00 00 00 00 00                           |........|
```

3) SSH keys: The Seahorse application provides also an additional feature that allows the user to generate, store, and track personal SSH keys. Since that kind of information is sensitive as well and might reveal more insights about the user's behavior we concentrated on extracting as much as possible data regarding the stored SSH keys.

We started our examination by searching for actual key entries and invoked *heapsearch* with their *names*. As a result, we got exactly one chunk per entry. It is referenced by a single container with a fixed size of 80 bytes (x32: 40). After investigating its content with *heaprefs*, we could conclude that it holds *five* pointers in total:

- **Offset: 0**: Contains the absolute path of the *public* key on the user's file system. It is terminated by null bytes.
- **Offset: 16 (x32: 8)**: Contains the absolute path of the *private* key on the user's file system (e.g., `/home/user/.ssh/id_rsa`). It is terminated by null bytes.
- **Offset: 24 (x32: 12)**: Leads to a chunk with a various size that holds the entire public key (ASCII-armored) as a string. It is terminated by null bytes.
- **Offset: 32 (x32: 16)**: Holds the already mentioned name of the key.
- **Offset: 40 (x32: 20)**: Refers to a chunk that contains the *fingerprint* of the current key entry. An exemplary output is given in Listing 8. It has a size of 80 bytes (x32: 72), the string starts at offset 0, and is terminated by null bytes.

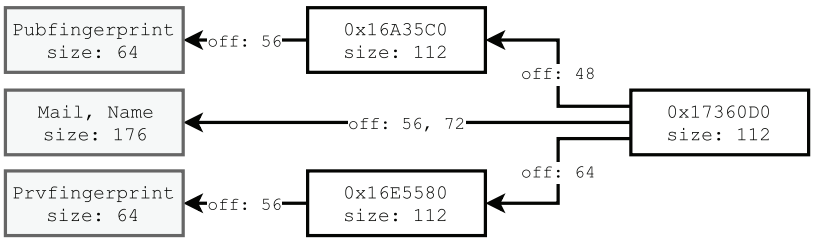

Figure 8: Illustration of the developed structure to retrieve the information regarding a PGP key.

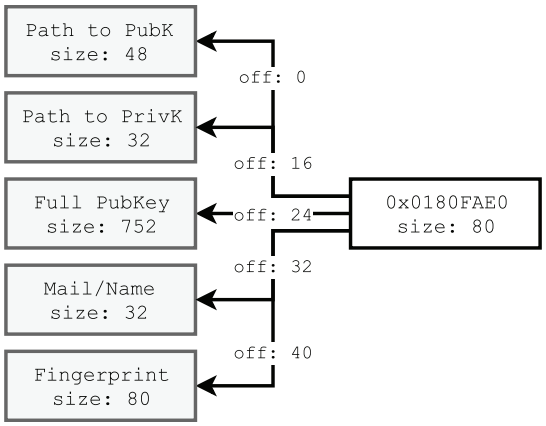

Figure 9: Illustration of the developed structure to retrieve the information regarding a SSH key.

Listing 9: Resulting output of the *seahorse* plugin. All previously gathered information are printed in a table.

```
entry          name          type                          content
------ ------------------ ------------- ------------------------------------------------
Name of password entries:
  1     github             Stored Note
[...]
PGP keys:
  1     hans.w@exam.com
  1.1                      Mail          hans.w@exam.com
  1.2                      Name          Hans Wurst
  1.3                      Note          test
  1.4                      Priv-SHA      3089E99B1599C2E894485B01231C331E48E854F6
  1.5                      Pub-SHA       66DD35661FE1695B92F5BBFD2DB18518A1A1F61F
[...]
SSH keys:
  1     test@test.com
  1.1                      Fingerprint   b1:fd:2b:9b:62:ba:f7:ec:44:a6:c2:20:b2:85:fa:58
  1.2                      Name          test@test.com
  1.3                      Path Private  /home/user/.ssh/id_rsa
  1.4                      Path Public   /home/user/.ssh/id_rsa.pub
  1.5                      Public Key    ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDcmCvR7Rrq
                                         [...]
                                         iw== test@test.com
```

Listing 8: Hexdump of a chunk containing the *fingerprint* of a SSH key.

```
00000000  62 31 3a 66 64 3a 32 62  3a 39 62 3a 36 32 3a 62  |b1:fd:2b:9b:62:b|
00000010  61 3a 66 37 3a 65 63 3a  34 34 3a 61 36 3a 63 32  |a:f7:ec:44:a6:c2|
00000020  3a 32 30 3a 62 32 3a 38  35 3a 66 61 3a 35 38 00  |:20:b2:85:fa:58.|
00000030  40 0e 7d 01 00 00 00 00  60 40 7e 01 00 00 00 00  |@.}.....`@~.....|
00000040  50 00 00 00 00 00 00 00                           |P.......|
```

Figure 9 concludes the here developed structure of a SSH entry. The container on the right holds the five pointers to the necessary information of the entry.

4) Plugin 3: `seahorse`*:* The plugin that recovers the above mentioned data structures is called *seahorse*. It is very similar to the one for gnome-keyring-d but has slight differences which we want to describe here. The output could again separated into three sections. The first one recovers all names of entries that are stored in currently open keyrings. Additionally, in the second section, the information regarding the stored PGP-keys is unveiled. For each entry the following that could be extracted: The *mail*-address and its corresponding *name*. Additionally, the describing *Note* if the user has entered one is printed. Furthermore, the plugin is able to provide the examiners the SHA-1 fingerprints of the private as well as the public key. In the last section, the plugin focuses on information regarding stored *SSH*-keys. For each entry, we are able to recover its *Fingerprint*, the *Name*, the location of the private as well as public key on the user's system, and the full public key itself.

(a) *Username* and *hostname*.

(b) *Src* and *Dest* IPv4 addresses.

Figure 10: Resulting structures to retrieve information from SSH's heap memory.

D. ssh

We examined the heap of ssh's process and focused on the following versions:

- 32-bit: version: OpenSSH-7.4.p1
- 64-bit: version: OpenSSH-7.4.p1

Unfortunately, We were not able to recover any fragments of the user's actual passphrase or the provided SSH-key. But we could at least recover several meta information regarding the current established connection.

1) Username and Hostname: Since we generated several scenarios where we connected to a remote server via SSH, we searched the entire heap of the process for the actual *username*. A chunk with a size of 32 bytes (x32: 16) was discovered that holds the entire name as a string (see Listing 10, `mem_test`). As one could see, the string does not only contain the *username* but also the *hostname* seems to be concatenated after a `@`-character.

Listing 10: Hexdump of the chunk containing the *username* and the *hostname*

```
00000000  6d 65 6d 5f 74 65 73 74  40 63 30 6e 66 2e 64 65  |mem_test@c0nf.de|
00000010  00 00 00 00 00 00 00 00                           |........|
```

That chunk is referenced by a single chunk with a fixed size of 32 bytes (x32: 16) which includes two pointers: The first one at offset 0 leads to a chunk that holds the string `ssh`. Whereas the username is referenced by the second one at offset 8. Figure 10a concludes and depicts the here recovered structure.

2) IPv4 Addresses: Source and Destination: Since we created the test scenario, we know to which server the user is connected and which IP addresses are used. So, we searched the entire heap for further information regarding the connection.

The *heapsearch* plugins unveils exactly one chunk that contains the IPv4 address of the *destination* (see Listing 11). It has a size of 32 bytes, the string starts at offset 0 and is terminated by null bytes.

Listing 11: Hexdump of the chunk containing the *ip of the destination*

```
00000000  31 38 38 2e 36 38 2e 35  30 2e 38 00 00 00 00 00  |188.68.50.8.....|
00000010  00 00 00 00 00 00 00 00                           |........|
```

That chunk is referenced by a single chunk with a fixed size of 2144 bytes (x32: 1080). It contains *two* mentionable pointers. The first one is located at offset 16 (x32: 8) and refers to the above mentioned IP address. The second one (offset 32; x32: 16) leads to an additional chunk that holds the local *source* IPv4 address of the client (e.g., `192.168.0.12`).

The resulting simple structure illustrated in Figure 10b. The right chunk contains a pointer at offset 16 (x32: 8) to the destination's IPv4 address. Furthermore, it refers to a chunk with the local address (offset 32 resp. x32: 16).

3) Plugin 4: `ssh`: The plugin *ssh* is able to recover the above mentioned data structures. It is able to support the forensic examiners to get information of currently opened connections. The resulting output of the plugin is given in Listing 12. It consists of *five* columns in total. Besides the current *PID*, the *username* as well as the corresponding *hostname* is given. Additionally, the plugin is able to recover the IPv4 addresses of the source and the destination.

Listing 12: Resulting output of the *ssh* plugin. It prints the connection information consisting of username, local IP, hostname, destination IP.

```
  pid       username         source          hostname       destination
------ ---------------- --------------- ---------------- ---------------
 1074  mem_test         10.0.2.15       c0nf.de          188.68.50.8
```

E. sshfs

We examined the heap of the *sshfs*'s process. Therefore, we examined two different versions depending on the architecture:

- 32-bit: version: 2.8
- 64-bit: version: 2.8

Before we could start our analysis, several test scenarios were created to simulate a user's behavior and usage of the application. So we mounted actual directories on remote servers, with several different users and their corresponding credentials. Similar to the ssh process, we were not able to observe any appearance of the passphrase or at least fragments from it. Neither we could derive any assumptions of the used SSH key.

1) Username and Hostname: The *username* could be found in several chunks but only one could be used in order to extract that information reliably. Its content is shown in Listing 13. As one could see, it consists of the targeted *username* (here `user`), but additionally, the actual hostname is attached after a @-character (here `c0nf.de`). After running several scenarios we could verify that there is always such a chunk with a various length (depending on the user- and hostname). The string is terminated by null bytes and is encoded in utf-8. That chunk is referenced by a single chunk with a fixed size of 48 bytes where it is referenced at offset 40 (x32: 20).

Listing 13: Hexdump of the chunk containing the *username* and the *hostname*

```
00000000  75 73 65 72 40 63 30 6e  66 2e 64 65 00 00 00 00  |user@c0nf.de....|
00000010  00 00 00 00 00 00 00 00                           |........|
```

2) Mount Information: It could be helpful to get more information about *which* actual folder of the server is mounted and *where* it is located on the suspect's local machine.

We invoked *heapsearch* and passed the path of the local folder (e.g., `/home/user/Documents`). As a result, we got only one single chunk with a size of 64 bytes. The content is listed in 14. Besides the targeted mounting point, it also *could* but not necessarily *must* consist of the path of the actual remote folder (here `/home/user2`). After running with multiple scenarios, we could examine the following observations: Its size may differ depending on the actual path lengths. The string is follows a specific pattern: `<path-local-folder>\x00<path-remote-folder>`.

After searching for concrete references to that chunk, we have to admit, that *not* any pointer could be found. So it stays in the memory without any connection to further structures. That circumstance will make it more difficult in the actual plugin to identify that fragment because its size may differ depending on the length of the two paths.

Listing 14: Hexdump of the chunk containing the local folder path and the remote folder path.

```
00000000  2f 68 6f 6d 65 2f 75 73  65 72 2f 44 6f 63 75 6d  |/home/user/Docum|
00000010  65 6e 74 73 00 2f 68 6f  6d 65 2f 75 73 65 72 32  |ents./home/user2|
00000020  00 65 3a 2f 68 6f 6d 65  2f 75 73 65 72 32 00 00  |.e:/home/user2..|
00000030  00 00 00 00 00 00 00 00                           |........|
```

3) Filelist: As the last step, we want to examine more information that reveals specific user action and could, therefore, support the investigators to reconsider past events.

Our first attempt was to find a listing of *all* available files that are synchronized in that session. But after searching for concrete entries, we realized that just a modified version is present, which means: All file entries that are touched by the last command the user typed in the console are retrievable.

We observed for each file entry one single chunk whose content is shown in Listing 15. It has a size of 32 bytes. Its string starts at offset 0 and is terminated by null bytes.

Listing 15: Hexdump of the chunk containing a filename.

```
00000000  2f 67 65 74 5f 75 73 65  72 73 77 2e 73 68 00 00  |/get_usersw.sh..|
00000010  00 00 00 00 00 00 00 00                           |........|
```

All here examined entries are gathered by one single chunk which size may differ depending on the amount of entries. It references each string with a separate pointer but a specific pattern regarding the offset is not recognizable. That clustering chunk is also referenced by a container with a fixed size of 1024 bytes (x32: 528). The corresponding pointer is located at offset 120 (x32: 76). All other pointers could be marked as nonrelevant. Figure 11 illustrates the resulting structure.

4) Plugin 5: `sshfs`: The plugin is called *sshfs* and reveals the above mentioned data structures and prints them to the examiner's console. A exemplary output is given in Listing 16. Besides the *username* and *hostname* that are used to establish the connection to the remote file system, the plugin is also able to recover the actual mounted folder of the server (*folder_server*) and the corresponding mounting point of the local system (*folder_local*). Additionally, the above mentioned partial filelist is printed as a list.

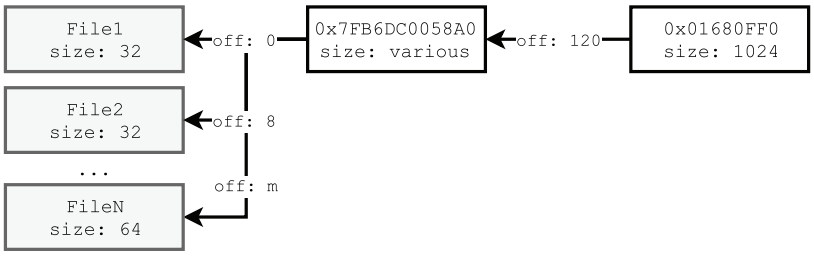

Figure 11: Illustration of the developed structure to retrieve a partial *filelist* that contains all recently touched files.

Listing 16: Resulting output of the *sshfs* plugin. It prints a table consisting of the columns *pid*, *entry*, *name*, *username*, *hostname*, *folder_server*, and *folder_local*.

```
 pid   entry         name         username    hostname     folder_server    folder_local
------ ----- ---------------- ----------- ------------- ---------------- ---------------
 1112    1   /                mem_test    c0nf.de       home/mem_test    /home/user/tmp
 [...]
 1112   33   /owntmp/..       mem_test    c0nf.de       home/mem_test    /home/user/tmp
```

F. pwsafe

We have to mention that the application could lock per default a previously opened password database after five minutes. So all here examined results including extractable information, and data structures are just available as long as the corresponding database is not locked. Additionally, the unencrypted *master password* that unlocks the entire database could not be found in the heap. The discovered structure and result of that section is given in Figure 12. We provide a detailed description of how we retrieved all shown insights in the following.

1) Password entries: After we realized that the master password is not present in the memory, we focused on the actually stored password entries that could contain describing fields like *Title*, *Username*, *Password*, *URL*, *Email*, and *Notes*.

We found for each title two potential containers: The first one has a size of 80 bytes and the string is encoded as 32-bit little-endian. The second one has a size of 32 bytes and its content is encoded as utf-8 (see Listing 17). As one could, the corresponding username is concatenated in square brackets.

Listing 17: Hexdump of an entry that contains only title and username.

```
00000000  4d 79 55 6e 69 20 5b 75  6e 69 73 68 6f 72 74 5d  |MyUni [unishort]|
00000010  00 00 00 00 00 00 00 00                           |........|
```

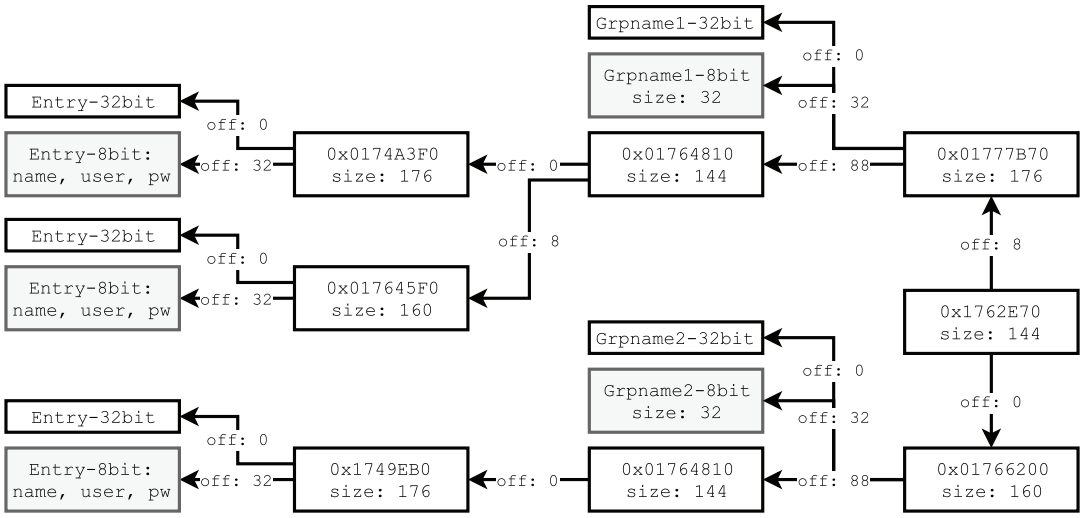

Figure 12: Illustration of the developed structure to retrieve all password entries and their corresponding group container.

Both chunks for each title are referenced by exactly one chunk. The two corresponding pointers are located at offset 0 (utf-32) and 32 (utf-8) (see also Figure 12). After examining that structure in different scenarios, we could assume that such a clustering chunk with its pointers always exists but its size may differ. These chunks are again referenced by chunks with a fixed size of 144 bytes that reference a different amount of such password entries. The pointers are aligned behind each other starting at offset 0. With the help of that we are able to recover *all* existing entries of that database.

The actual *password* could only be found in some special user behavior: At first, the user is able to copy a password to the clipboard. In that case, a chunk could be observed a short amount of time in the heap which is freed very quickly. In the second case, the user could set up the manager in a way that the password is displayed besides the title and the username in the application. That information is also concatenated to the already examined chunk that holds the title and the username. A hexdump of the so resulting string is given in Listing 18. As one could see in comparison to Listing 17, the substring `{secret123}` is concatenated.

Listing 18: Hexdump of an entry that contains the password.

```
00000000  4d 79 55 6e 69 20 5b 75  6e 69 73 68 6f 72 74 5d  |MyUni [unishort]|
00000010  20 7b 73 65 63 72 65 74  31 32 33 7d 00 00 00 00  | {secret123}....|
00000020  33 00 00 00 00 00 00 00                           |3.......|
```

2) Clustering and name of the group: In the above listing of an entry's elements, we did not mention the *Group*-field. It is used to structure the password entries.

We previously found out, that just a few chunks reference all password entries. Alone that could be seen as of structure, but we also want a conceptional proof that all these entries belong to one common group.

The name of the group is present in *two* chunks: The first contains the string as 32-bit encoded and the second one as 8-bit (see Listing 19). The size may differ depending on the length of the string. Those two chunks are referenced by one common container (offsets 0 and 32). Additionally, it has a pointer at offset 88 that leads to password entries (title, username) from the previous section. So that chunk is the connection between group name and the entries that are stored within that group (see Figure 12). That assumption is verified after running several test scenarios.

Finally, we recognized that these group containers are referenced by one single chunk with a fixed size of 144 bytes. It contains as many pointers as group are present which are aligned behind each other starting at offset 8.

Listing 19: Hexdump of the chunk containing the name of the group.

```
00000000  53 63 68 6f 6f 6c 00 00  38 1b 3e 05 66 7f 00 00  |School..8.>.f...|
00000010  20 00 00 00 00 00 00 00                           | .......|
```

3) Plugin 6: `pwsafe`: The here presented data structures are recoverable with the help of our plugin called *pwsafe*. Listing 20 illustrates its resulting output on the examiner's console. It is a table that consists of *five* columns in total. All entries of opened password databases are recoverable. For each entry, the plugin unveils its corresponding group where the password is stored. Additionally, the *title* and the *username* is recoverable. If the user has a setup its user interface in a way, that the password is also displayed every time, we are able to recover that entry as well in clear.

Listing 20: Resulting output of the *pwsafe* plugin. A list of all password entries is printed including the *group*, *title*, *username*, and *password* (if present).

```
  entry   group          title               username             password
  ----- ---------- -------------------- -------------------- --------------------
Task: pwsafe (1198):
    1   Personal   Facebook             hans.wurst           ananas
[...]
   42   School     MyUni Copy # 9       unishort             secret123
```

G. SQLite

We examined two different versions depending on the architecture:

- 32-bit: version: 3.16.2
- 64-bit: version: 3.17.0

1) Command History: Before we could start the analysis, we have to create a test scenario where we executed several commands (e.g., *.tables* and *.help*).

After we searched the entire allocated chunks for concrete entered commands, we recognized that each command is stored in one separate chunk. Its size may differ depending on the actual length. The string starts at offset 0 and is terminated by null bytes and has at least a size of 32 bytes (x32: 16). Each container is referenced by an additional chunk with a fixed

size of 32 bytes (x32: 16). With the help of *heaprefs* (see Figure 13), we recognized its two pointers. The one at offset 0 leads to the entered command string. The second one at offset 8 (x32: 4) refers to another container whose output is given in Listing 21. It contains a UNIX-timestamp which represents the time and date the command was executed. The stamp starts by a one-byte offset and ends with null bytes. As a connection between all entries, we identified one chunk with a size of 4032 bytes (x32: 2016) that contains pointer to each entry. By traversing that chunk, we are able to recover the whole history.

Listing 21: Hexdump of the chunk containing the password.

```
00000000  00 31 35 30 34 30 30 31  39 38 32 00 8e 7f 00 00  |.1504001982.....|
00000010  09 09 49 42 4d 35 33 34                           |..IBM534|
```

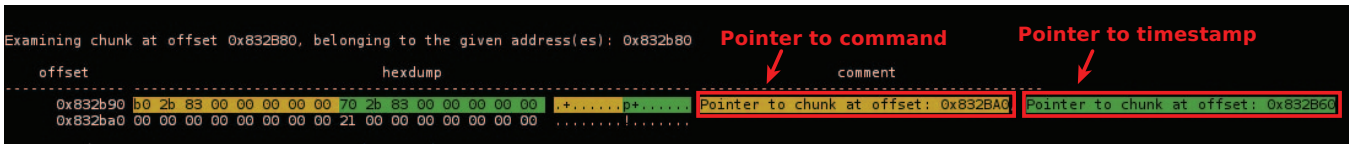

Figure 13: Analyzing a chunk's references by calling *heaprefs*.

Figure 14 illustrates a minimized example of the here developed data structure. A large chunk refers to all entries of the command history (here at offset 32, and 40). Each container has a reference to the actual string and the corresponding timestamp.

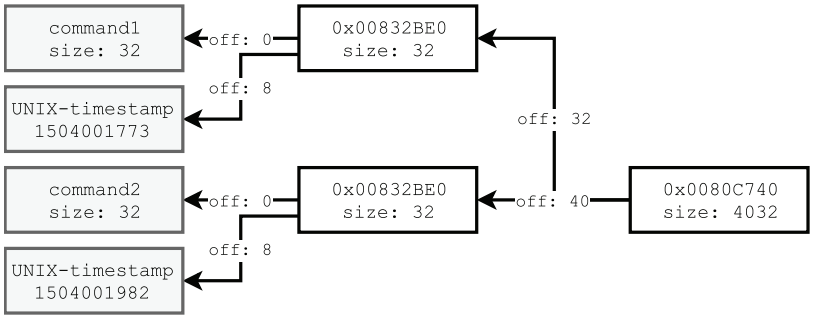

Figure 14: Sqlite: Minimized illustration of two entries of the command history.

2) Database Schemas: We searched the memory for the actual name of table using the *heapsearch* plugin. As a result, and after some further examination, we got precisely *one* container that holds the name of a database. Its size may differ depending on the length of the name. It is terminated by null bytes and starts at the beginning of the chunk's data part. That chunk is referenced by a additional container with a fixed size of 144 bytes (x32: 96). With the help of *heaprefs* we could identify *two* useful pointers in total. The first one is located at offset 0 (x32: 8) and leads to the already examined name of the table. The second pointer is located at offset 8 (x32: 12) which leads to a chunk with a various size. That container includes a different amount of pointers: Starting at offset 0, at every 32 bytes (x32: 16) is a reference that leads to another string (see Listing 22).The content has the following pattern: `<field-name>\x00<field-type>\x00\x00`. The name of the field starts at byte 0 and is terminated by a null byte. Followed by the type of the field which is also finalized by null bytes. By traversing the existing references, whereas the end is reached by finding a non-valid pointer (mostly null-Pointer), we are able to recover the complete database scheme of the actual table.

Additionally, we found a sort of linked list that connects all schemas of the currently loaded database. The already mentioned chunk with a size of 144 bytes are referenced by separate chunks with a fixed size of 48 bytes (x32: 32). Its pointers at offset 0 (x32: 8) and 8 (x32: 12) could be seen as the *prev* and *next* pointers of the linked list. The beginning or end is reached if corresponding pointer is null.

Listing 22: Hexdump of the chunk containing one column of a table.

```
00000000  69 64 00 69 6e 74 65 67  65 72 00 d2 8e 7f 00 00  |id.integer......|
00000010  28 f2 86 00 00 00 00 00                           |(.......|
```

Figure 15 illustrates the here developed data structure. All entries are accessible by traversing a linked list.

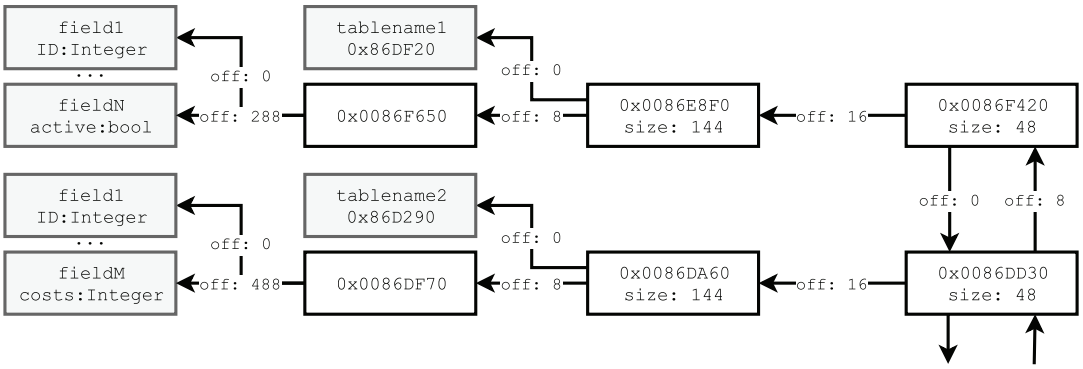

Figure 15: Illustration of the developed structure to retrieve the available tables together with their scheme.

3) Plugin 7: `sqlite`: The resulting plugin is called *sqlite* and its output is given in Listing 23. It could be separated into two sections. The first one prints the history of commands to the examiner's console including the corresponding timestamp of each entry. In the second section, we provide the examiner the complete scheme of all available/loaded tables within the database.

Listing 23: Resulting output of the *sqlite* plugin. At first the command history is printed. Followed by the name and schema of each available table.

```
   pid   entry         time                 command
  ------ ----- -------------------- ------------------------
   1262    1   2017-08-29 10:16:09Z  quit
[...]
   1262    6   2017-08-29 10:19:42Z  .tables

Table 1: auth_group
           1   id                   integer
           2   name                 varchar(80)

Table 2: djcelery_workerstate
           1   id                   integer
           2   hostname             varchar(255)
           3   last_heartbeat       datetime
[...]
```

H. OwnCloud

In the following section we want to provide deeper insights about the analysis of owncloud's process. Therefore, we examined two different versions depending on the architecture:

- 32-bit: version: 2.3.4
- 64-bit: version: 2.3.2

1) Credentials: The password of a user is stored in a single chunk with a different size depending on the actual length of the phrase. The string is encoded as 16-bit littleendian (UTF-16) and is listed in 24. It starts after 24 bytes and is terminated by null bytes. That chunk is again referenced by a container with a size of 96 bytes (x32: 48). The corresponding pointer is located at offset 32 (x32: 16).

But that container has another pointer at offset 24 (x32: 12). With the help of that, we were also able to identify the corresponding *username*. Essential was to find a concrete connection between the name and its passphrase. The string is shown in Listing 25. As one could see, it is again encoded in UTF-16, starts after 24 bytes and is terminated by null bytes.

Since the size of 96 bytes is very common, we need to retrieve more characteristics in order to identify that structure uniquely under all available chunks. The above mentioned container is referenced by a single chunk with a fixed size of 192 bytes (x32: 72) that holds a pointer at offset 152 (x32: 4) to the target container. It is also referenced back from it at offset 16 (x32: 4).

Figure 16 shows the here developed structure for extracting a pair of credentials from the memory of ownCloud's process. By searching the entire memory of the process, we could identify *all* available accounts.

Listing 24: Hexdump of the chunk containing the password.

```
00000000  02 00 00 00 0c 00 00 00  14 00 00 00 01 00 00 00  |................|
00000010  18 00 00 00 00 00 00 00  6d 00 65 00 6d 00 5f 00  |........m.e.m._.|
00000020  70 00 61 00 73 00 73 00  77 00 6f 00 72 00 64 00  |p.a.s.s.w.o.r.d.|
00000030  00 00 9a 01 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00                           |........|
```

Listing 25: Hexdump of the chunk containing the username.

```
00000000  05 00 00 00 08 00 00 00  14 00 00 00 01 00 00 00  |................|
00000010  18 00 00 00 00 00 00 00  6d 00 65 00 6d 00 5f 00  |........m.e.m._.|
00000020  74 00 65 00 73 00 74 00  00 00 6d 01 00 00 00 00  |t.e.s.t...m.....|
00000030  d0 48 90 01 00 00 00 00  00 00 00 00 00 00 00 40  |.H.............@|
00000040  52 00 00 00 00 00 00 00                           |R.......|
```

2) Hostname: Since the login credentials are useless without the corresponding *hostname*, we discovered several chunks that contain exactly that information. The results could be minimized to one concrete chunk which size may differ. The string starts with `Connected to` and contains the username as well as its corresponding *hostname*. It is encoded as UTF-16 and is terminated by null bytes. That chunk is referenced by a container with a fixed size of 640 bytes (x32: 408) which holds a pointer at offset 504 (x32: 328) to the target string. By following that pointer and knowing the pattern of the string where are able to identify the hostname uniquely.

3) Synchronization protocol: The investigators might need to know something about what actions the user had taken before the memory was acquired or, in the case of ownCloud, what files were deleted or at least modified in that session. That information is provided by the *synchronisation protocol*.

For each synchronized file, we recognized a single chunk that holds besides the targeted *filename* also the entire *relative* path on the user's system (e.g., `ownCloud_Manual.pdf` is realized as `Documents/ownCloud_Manual.pdf`). The string is encoded as UTF-16 and is terminated by null bytes. The container has a different size depending on the length of the path. It is referenced by a chunk with a fixed size of 32 bytes where the corresponding pointer is located at offset 0. That chunk is again connected to a container with a fixed size of 80 bytes which holds *four* interesting pointers in total that allows us to rebuild an entire entry of the protocol:

1) **Offset: 16**: Points to a chunk that references a string that represents a timestamp when the last action was taken (see Listing 26).
2) **Offset: 24**: Refers to the above mentioned relative path of a file entry.
3) **Offset: 32**: Points to a chunk that references a string that represents the actual root folder of the owncloud.
4) **Offset: 40**: Points to a chunk that references a string that represents the last taken action (e.g., *Deleted*, *Downloaded* etc.).

We were not able to find any connection between different entries like a list or similar. So we need at least a way to identify the above structure uniquely. The above presented gathering container of an entry is referenced by an additionally

Figure 16: Illustration of the developed structure to retrieve the username and password of an owncloud account.

Listing 26: Hexdump of the chunk containing a timestamp.

```
00000000  01 00 00 00 11 00 00 00  14 00 00 00 02 00 00 00  |................|
00000010  18 00 00 00 00 00 00 00  31 00 36 00 2e 00 30 00  |........1.6...0.|
00000020  37 00 2e 00 31 00 37 00  20 00 31 00 39 00 3a 00  |7...1.7. .1.9.:.|
00000030  34 00 34 00 3a 00 32 00  38 00 00 00 64 00 00 00  |4.4.:.2.8...d...|
00000040  50 00 00 00 00 00 00 00                           |P.......|
```

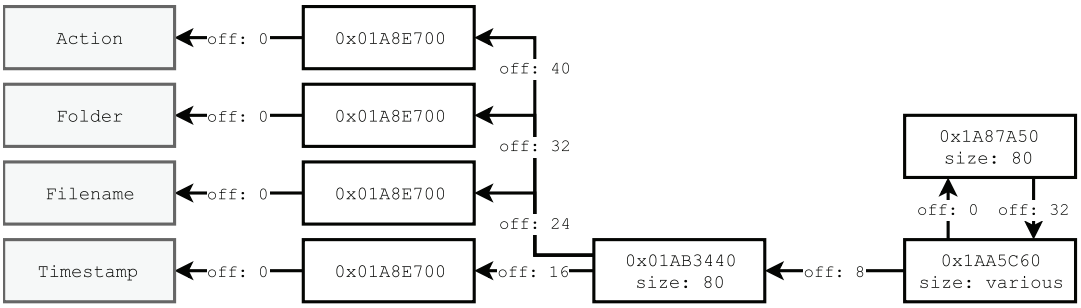

Figure 17: Illustration of an entry of the synchronisation protocol.

chunk with various size at offset 8 (x32: 4). Additionally, that chunk has a pointer at offset 0 to another chunk with a fixed size of 80 bytes which includes also a pointer back at offset 32 (x32: 16). Figure 17 illustrates that chain with corresponding offsets. Furthermore, it concludes the here developed data structure for an entry of the synchronization protocol.

4) Plugin 8: `owncloud`: The plugin that recovers the above mentioned data structures is called *owncloud*. With the help of that, the examiner is supported by retrieving useful information to track the open connections to owncloud servers as well as actual user interaction by providing the latest synchronisation protocol of the user's actions. The plugin could be divided into two sections. The first one prints the recoverable information data. That includes for each connection the actual *hostname* and the corresponding pair of *username* and *password*. So the examiner is able to investigate the cloud storage separately by connecting with these credentials. In the second section, we provide the synchronisation protocol of the current session. That includes for each touched entry, the *name*, a *timestamp*, the corresponding *folder*, and the taken *action* (includes *Downloaded*, *Deleted*, etc.).

Listing 27: Resulting output of the ownCloud plugin. After the header, it prints all found login credentials including the *hostname*, *username*, and the *password*. Followed by the synchronization protocol.

```
Hostname: https://cloud.c0nf.de
Username: mem_test
Password: mem_password
--------------------------------
entry         time                    file                folder      action
----- -------------------- -------------------------------- ------------ ----------
1     2017-07-16 19:44:28  ownCloud Manual.pdf              ownCloud      Downloaded
2     2017-07-16 19:44:25  Documents/Example.odt            ownCloud      Downloaded
[...]
```

IV. Evaluation

This section covers the evaluation results for all developed plugins. All tests have been performed in the following environments:

- Debian "stretch" 32 bit, x86, Kernel Version 4.9.30-2+deb2u5
- Arch Linux 64 bit, x64, Kernel Version 4.4-66
- Glibc Versions: 2.24, 2.25

The current set of plugins covers a specific set of Linux user space applications with a specific version. Hence, the evaluation concentrates on these versions in the described environment and can not make a definite statement for other settings.

A. Correctness

In this section, we describe the evaluation of the developed plugins and their resulting output for correctness:

1) Password managers: gnome_keyring, seahorse, pwsafe: During our work, we focused on *three* different password managers and analyzed their available data and structure in the heap. We divided this section regarding the actual data that is recoverable.

Password entries: The main functionality of our tools covers the scenario where the keyrings resp. databases are open/unlocked. The following special cases are covered within our evaluation:

- **Amount of entries**: It is important that all available entries of the currently open keyrings/databases are recognized. To verify that behavior, we created several test cases where we started by one single entry and increased the number to 200. All artifacts were found.

- **Length of strings**: That includes the actual *name* of the stored notes and in the case of *pwsafe* also the *username*. To verify different lengths of those strings, we created user scenarios where the *note/name* and *username* starts from 4 characters and increases to 48. All different manifestations were recognized by our tools.

Another special example is given by *pwsafe*. In the actual application, the user is able to display the *password* besides the current *name* and *username* of the entry. That is also covered by the above mentioned scenarios.

PGP information: Seahorse allows as the only password manager to store information regarding PGP keys. We expect from the plugin to identify the following special cases:

- **Multiple entries**: For an ongoing investigation it is important that *all* stored PGP-entries are gathered by our plugin. In order to evaluate that behavior, we applied several scenarios. At first, only *one* entry is stored within the application. That amount is increased up to 32. All manifestations are identified and printed to the examiner's console.
- **Correct mapping**: Since we recognized multiple accounts, we also need to verify that the extracted information are mapped together correctly. Therefore, we created several tests where we added *two* accounts with different *name* and *fingerprints* and compared the produced output of our tool with the expected account information. The number of accounts was also increased up to 32.

SSH information: The last kind of data that can be managed by *Seahorse* and *gnome_keyring_d* (gkr) are SSH-keys and their corresponding meta-information. The following scenarios were applied in order to verify the functionality of the plugins:

- **Multiple entries**: Multiple accounts were added, starting from *one* up until 32. In the case of *gnome_keyring* we checked, whether *all* entries are identified and printed correctly. Additionally, we compared the meta-information extracted by the *seahorse* plugin with our initial input. No discrepancies could be observed.
- **Correct mappings**: Regarding PGP-keys with *seahorse*, we created several test files where we added *two* up until to 32 SSH accounts and checked whether the developed structures are discovered correctly, which means that the key of account A correlates with the expected other fields (e.g., name, fingerprint, etc.).

2) cUrl: The process of *cUrl* is very short-living and though it is more a proof of concept plugin since all data is lost when it is terminated.

- **Length of URL and output filename**: To verify, that all kinds of strings are recognized by our tool, we created scenarios, where the URL and filename are very short (*six* characters) and increased it up to 96. We compared the expected results with the ones the plugin extracted. All scenarios were recognized correctly.
- **Presence of login credentials and filename**: The user has optionally the possibility to supply credentials via command line or within a file. To evaluate our plugin in this case, we created two kinds of scenarios: In the first one, we used the application without any provided credentials. In the second one, we called the cUrl passing the username and entered the password afterwards. The plugin has to recognize whether such information are provided or not and if so, are those information correct.
- **Length of username and password**: Additionally to the previously mentioned test, we used various lengths for the username and password The strings started at a length of $four$ characters and were increased up until 48.

3) ssh: For this plugin, we focused on the following cases:

- **IPv4 addresses**: The plugin uses a regex to verify the validity of the potentially identified IP addresses. If the address seems invalid, the output is marked accordingly.
- **Length of user- and hostname**: In order to provide a wide range of reliable results, we created several test scenarios to verify different lengths of the username and the hostname, where the length of the username starts from a minimum of 4 characters and increases to 48. Additionally, the hostname is expanded up to 64 characters. All scenarios are processed correctly by our plugin.

4) sshfs: We've already mentioned the dynamic behavior of the resulting filelist for this plugin. The number of entries depends on the last command the user has entered on the mounted filesystem. For example, a `ls` command touches all files in the current folder whereas the `vim <filename>` touches only one particular file, which influences directly the entries of the filelist.

To verify the correctness, we created these scenarios:

- **Length of strings**: This includes the *username*, the *hostname* and the entered *filepaths*. In order to verify the correct results of our plugin, we created several test scenarios where we varied the strings from short (six characters) to longer ones (up to 96 characters). In all scenarios, the plugin produced the expected results.
- **Length of filelist**: Regarding the filelist result, we simulated user behavior with several commands such as listing directory content and changing single and multiple files/directories.

We recognized during our evaluation, that the mount information regarding the local and especially the remote folder is not that easy to recover since the data is stored in a chunk, which is not referenced by any other container and therefore, we try to identify this chunk via its size. Another issue is the presence of the remote folder. While the local folder was always present in our evaluation, the remote folder is in the 32-bit application not discoverable and might be missing in the output.

5) sqlite: The *sqlite* plugin provides information regarding the commands a user has entered and is also capable of recovering the database scheme for each loaded table (see Listing 23). Besides the already mentioned basic functionality, we evaluated the plugin for the following cases:

- **Amount of entered commands**: Essentially for a reliable output is the correct number and content of all entered commands. We verified that behavior by creating several test scenarios with a different amount of commands beginning from *five* to 100.
- **Timestamp**: Additionally to the extracted command string, the corresponding timestamp should match as well. The setup is almost the same and the expected stamp is compared with the extracted one.
- **Complete scheme**: The tool should be able to rebuild the complete scheme for each available table. To verify that behavior, the first scenario contains *one* table with just *two* fields. As expected, the plugin was able to print that information correctly. The number of tables and fields were increased individually up to a number of 32 fields per table. Each scheme was identified and printed correctly.
- **Amount of loaded tables**: The above mentioned test scenarios were also used in order to verify that *all* loaded tables are unveiled by our plugin. The number of available tables started with *one* and increased up to 128. The plugin recognized each table correctly.

6) Cloud Storage: owncloud: The following scenarios were tested for this plugin:

- **Multiple accounts**: It is important that *all* available usernames and their corresponding passwords are extracted by the plugin, including the *hostname*. To verify that behavior, we created several test files where multiple accounts were present. Starting with a total number of *two* accounts and ending at 32.
- **Long username/password/hostname/filename**: The user could e.g. use extra long usernames or passwords. To verify that each case of long strings is processed correctly, we created various scenarios with long strings for each entry. We entered for example passwords consisting of *four* characters and ended at a length of 128 which covers the default configuration of the server.
- **Complete Synchronisation Protocol**: We also had to verify, whether the extracted protocol contains the same entries as the one on the suspect's system. Therefore, we created scenarios, which contains a different amount of entries (*one* up to *255*) to cover the most use-cases for our evaluation. The plugin extracted all manifestations.
- **Correct protocol entries**: Additionally, we verified our output in a way that each extracted entry of the protocol is correct and reflects the actual protocol of the user's process. Therefore, we compared the extracted entries with the expected values.

B. Performance

Besides the already described *functionality* of our tools, we evaluated them also in regards to their *performance*. The measurements are recorded on a system with a Intel Core i5-4670 quad-core CPU with 3,400 MHz, 16 GB RAM, and Linux Debian 9 64-bit OS. To get a more precise result, we measured the execution *five* times each and calculated the average duration. Furthermore, we ran each plugin on a 32 and 64 bit memory dump. Most of the plugins have a runtime of under *ten* seconds until they print out their results to the examiner's console. For example, the *ssh* plugin took 2.204 seconds (32-bit) and 2.229 seconds (64-bit). There are no mentionable differences between the 32-bit and the 64-bit version of a plugin since their results are nearly the same. There are *three* plugins whose time of execution is significantly greater than the others. Those plugins are *seahorse*, *pwsafe* and *owncloud*. For example, the measured time of *seahorse* is 79.576s whereas the one of *gnome_keyring* is just 4.096s. The reason for those peaks can be explained by the higher amount of chunks in these applications that have to be processed. The higher amount is most probably a result from the graphical user interface. Seahorse for example had in our test scenario a total number of 61,523 allocated chunks, whereas gnome-keyring had only 3,148.

V. Conclusion

All in all we conclude that the heap is a source of forensically valuable artifacts, which are extractable with the right tools. By using the plugins of Block [9], we were able to recover several existing data structures for each selected user space application, which enabled us to make a statement about *what* data is present in the memory, *where* it is located and *how* it is structured.

We adopted those insights and developed plugins for the Rekall Memory Framework for each analyzed application, which we publicly release alongside with this paper. With the help of our plugins, the forensic examiners are able to extract valuable information from a suspect's memory without any deeper knowledge about the data located in the applications' heap. All plugins provide their result in a human readable and comprehensible table and are designed in a way that allows an easy extension or adjustment for future changes due to new application versions. This is accomplished by storing all data offsets, sizes and constants in own container classes.

A. Limitations

The current state of our set of plugins is fully functional and can be used in forensic investigations in order to extract relevant artifacts from the suspect's memory such as *command history* and *login credentials*.

Currently, our plugins support officially only the mentioned versions and might fail on new/older versions, if e.g. some fields in the data structures change. This might also lead to unreliable results as long as the plugins are not updated.

The *performance* of our plugins on the other hand gets worse with an increasing amount of chunks. As can be seen in our evaluation, especially applications with a graphical user interface (such as *Seahorse*) allocate an enormous number of chunks and consequently, the actual runtime increases. It remains as future work to evaluate if there is a way the detect and exclude non-relevant junks stemming from the graphical user interface, or if there are other ways to furhter improve runtime performance.

B. Future Work

To increase the version support, on the one hand older versions of the target applications should be analyzed, and on the other hand, new versions, as soon as they are released. Due to the plugin design, most changes resulting from different versions, such as different sizes and pointer offsets, should be easily applicable.

Regarding the current set of plugins, some of them could be improved to provide more valuable information. The current state of the *gnome-keyring-d* for example (see Section III-B) gathers the present SSH private keys, but the connection to a user account or the public key is missing so far.

To cover a broader range of Linux user space applications, the current set of Rekall plugins should be expanded. Therefore, further applications that depend on the Glibc heap implementation need to be identified and their memory analyzed, in order to recover existing data structures and hence, implement new plugins.

REFERENCES

[1] A. Case, L. Marziale, C. Neckar, and G. G. Richard, "Treasure and tragedy in kmem_cache mining for live forensics investigation," *Digital Investigation*, vol. 7, pp. S41–S47, 2010.

[2] A. Aljaedi, D. Lindskog, P. Zavarsky, R. Ruhl, and F. Almari, "Comparative analysis of volatile memory forensics: live response vs. memory imaging," in *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*. IEEE, 2011, pp. 1253–1258.

[3] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[4] M. Cohen, "Rekall memory forensic framework," http://www.rekall-forensic.com/, [Online; accessed 16-May-2017].

[5] The Volatility Foundation, "Volatility," http://www.volatilityfoundation.org/, [Online; accessed 16-May-2017].

[6] M. Cohen, "Forensic analysis of windows user space applications through heap allocations," in *Computers and Communication (ISCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 237–244.

[7] Google Inc., "Rekall: Extract command history," http://www.rekall-forensic.com/docs/Manual/Plugins/Windows/#cmdscan, [Online; accessed 16-May-2017].

[8] Google Inc.., "Rekall: Scan the bash process for history," http://www.rekall-forensic.com/docs/Manual/Plugins/Linux/#bash, [Online; accessed 16-May-2017].

[9] F. Block and A. Dewald, "Linux memory forensics: Dissecting the user space process heap," *Digital Investigation*, vol. 22, pp. P66 – P75, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287617301895

[10] S. L. Garfinkel, "Digital forensics research: The next 10 years," *Digital Investigation*, vol. 7, pp. S64–S73, 2010.

[11] M. Valersi, "Analyse und sicherung semi-persistenter spuren im hauptspeicher des firefox nach private browsing sitzungen," Master's Thesis. https://www1.cs.fau.de/filepool/gruhn/MDF_M18_2015-11-09_Michael_Valersi.pdf, [Online; accessed 15-October-2017].

[12] Free Software Foundation, "The gnu c library," https://www.gnu.org/software/libc/, [Online; accessed 16-May-2017].