Digital Investigation 22 (2017) S66–S75

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 USA — Proceedings of the Seventeenth Annual DFRWS USA

Linux memory forensics: Dissecting the user space process heap

Frank Block [a, c, *], Andreas Dewald [b, c]

[a] ERNW GmbH, Heidelberg, Germany
[b] ERNW Research GmbH, Heidelberg, Germany
[c] Friedrich-Alexander University Erlangen-Nuremberg (FAU), Germany

Keywords:
Linux
Heap
Memory forensics
Glibc
Rekall

A B S T R A C T

The analysis of memory during a forensic investigation is often an important step to reconstruct events. While prior work in this field has mostly concentrated on information residing in the kernel space (process lists, network connections, and so on) and in particular on the Microsoft Windows operating system, this work focuses on Linux user space processes as they might also contain valuable information for an investigation. Because a lot of process data is located in the heap, this work in the first place concentrates on the analysis of Glibc's heap implementation and on how and where heap related information is stored in the virtual memory of Linux processes that use this implementation. Up to now, the heap was mostly considered a large cohesive memory region from a memory forensics perspective, making it rather hard manual work to identify relevant information inside. We introduce a Python class for the memory analysis framework Rekall that is based on our analysis results and allows access to all chunks contained in the heap and their meta information. Further, based on this class, six plugins have been developed that support an investigator in analyzing user space processes: Four of these plugins provide generic analysis capabilities such as finding information/references within chunks and dumping chunks into separate files for further investigation. These plugins have been used to reverse engineer data structures within the heap for user space processes, while illustrating how such plugins ease the whole analysis process. The remaining two plugins are a result of these user space process analyses and are extracting the command history for the *zsh* shell and password entry information for the password manager *KeePassX*.

Introduction

As the memory represents the current state of a running system, it contains for example information about browser history, entered commands, active network connections, loaded drivers as well as active processes and hence allows detailed insights into previous activities (Ligh et al., 2014). While much of this information is located in the kernel space and can be examined with existing solutions for various operating systems such as Rekall (Google Inc, 2016c) and Volatility (The Volatility Foundation, 2016), there is also a lot of information located in the user space that might be of interest in a forensic investigation, too. The heap of a user space process for example is typically a rich source of various kinds of data and, depending on the concrete application, might contain credentials, IP addresses/DNS names or a command history. However, this information is, at least in the context of Linux, not yet easily extractable.

To efficiently and reliably identify and extract this information, the investigator requires a view of the heap that is the same or at least similar to the one the process has: Knowledge about where the data is located, what kind of data is stored at a specific position and which amount of memory a specific data portion occupies. Otherwise, the investigator can only work with the heap as one large memory region, with all pieces of information located inside without any known structure.

Motivation

In the context of Linux processes, the focus of user space analysis was in the past limited to searches for specific patterns within the complete process memory or the whole heap/stack. For example, in order to reconstruct the command history for the Linux *bash* shell, Rekall's *bash* 7 plugin searches the heap for a hashtag followed by a Unix timestamp in string format (e.g. #1471572423) (Ligh et al.,

* Corresponding author. ERNW GmbH, Heidelberg, Germany.
E-mail addresses: fblock@ernw.de (F. Block), adewald@ernw.de (A. Dewald).

http://dx.doi.org/10.1016/j.diin.2017.06.002

2014, 630 ff.). If however the information of interest is not marked in an easily detectable way, a simple pattern matching will fail.

When taking a scenario in which the investigator identifies a certain string in memory and tries to identify references to it, this search might fail even though there are (indirect) references. This could be the case if the string is part of a struct or object and is not located at the beginning, while the pointer of interest references the struct instead of the string directly (see Section KeePassX for an example). To be able to find those references, the beginning of that struct must be known, which requires knowledge about the size of its fields and the location of the string within that struct. However, these details are normally not available in a black box analysis.

Contributions

In this work, we make the following contributions:

- We analyzed the Glibc heap implementation and summarize the information that enables an investigator to perform a manual heap analysis or implement his or her own tool for this purpose. In particular, we explain how heap structures are arranged and where they are typically located in memory.
- We demonstrate that some chunks might hide somewhere in the memory, for which we propose an algorithm to retrieve them.
- These insights have been used to develop a Python class called *HeapAnalysis*, which can be used to implement specific heap analysis plugins. Based on this class, we developed plugins that support the investigator in analyzing the heap and its chunks.
- We explain how data of user space processes can be analyzed by applying these plugins to gather relevant information (similar to the analysis done by Cohen (2015) for a Windows user space process).
- A result from this analysis are two further plugins: The first one gathers all executed commands from the heap of a *zsh* shell (Version 5.2) process and the second extracts the title, username, URL and comment field of all retrievable password entries from the heap of the password manager *KeePassX* (Version 0.4.3).

The *HeapAnalysis* class and all mentioned plugins support $\times 86$ and $\times 64$ architectures.

Alongside with this paper, we publish a technical report that contains further technical details and code listings (Block and Dewald, 2017).

Outline

This work is structured as follows: Section Glibc analysis covers details about the heap of user space processes that use Glibc's heap implementation. In Section Plugin implementation, we provide an overview of the developed heap analysis plugins, Section Evaluation covers the evaluation of those plugins, Section Application on real world scenarios provides a detailed analysis of applications, and Section Conclusion and future work concludes this paper.

Related work

Cohen (2015, p. 1138) states that “the analysis of user space applications has not received enough attention so far”. This not only underlines the motivation of this work, but also the reason for which there is not much literature about that specific topic. Existing literature in the field of Linux memory forensics mainly covers kernel related topics such as the work of Urrea (2006), Case et al. (2010) and Ligh et al. (2014).

The few exceptions are the work by Leppert (2012) and Macht (2013), who both focused on the Android operating system of mobile devices, which is Linux based, and analyzed applications and their heap data. However, their analysis concentrated primarily on serialized Java objects contained in the heap and not on the way heap objects are managed. Other exceptions are the already existing plugins *cmdscan* (Google Inc, 2016b) and *bash* (Google Inc, 2016a), which extract the command history from Windows' *cmd* and Linux's *bash* shell, respectively. These plugins, however, leverage the fact that in those cases it is possible to identify the information by only looking at the heap as one large memory region. Another related work is the analysis of Notepad's heap (Ligh et al., 2014, p. 223). This is to the best of our knowledge the only example that uses any heap details and, as most of the prior work, is related to Windows too.

Outside the scope of forensics, there has been fundamental research on the heap and, in particular, on the heap of Linux processes and how it is managed. Especially the research by Ferguson (2007) serves a solid understanding about Glibc's heap implementation. This previous research, however, focused more on the ways the heap can be exploited and hence does not provide enough information to reliably gather all relevant information from the heap in a memory forensics scenario.

The work of Cohen (2015) is the first to approach this research gap with a set of analysis tools for the Windows Operating system. While there are some similarities between Windows' heap implementation and the one from Glibc (for example in both cases an allocated chunk is preceded by a struct containing at least the chunk's size), they differ in the details.

Glibc analysis

In this section, we present the most important results of our analysis of the Glibc heap implementation from a memory forensics perspective (more detailed information can be taken from our technical report (Block and Dewald, 2017)).

Different heap implementations

The implementation examined in this work is Glibc version 2.23 (Free Software Foundation Inc, 2016), which is based on Wolfram Gloger's *ptmalloc2* (Gloger, 2006), but there are various other heap implementations, most of them used in the context of a certain operating system or application. Application developers can also decide to implement their own or use another, existing heap implementation (e.g. the case with *Mozilla* products such as *Firefox*). Such processes might then however not be analyzable using the information or tools introduced in this work.

Glibc heap overview

This section provides a high level overview of the most important objects and structs used in Glibc's heap implementation. Fig. 1 shows a potential heap layout of a running process, with a focus on references between the various elements. Starting from the lowest and most important level, a chunk contains the actual user/process data, which has been allocated e.g. explicitly via a malloc call or implicitly via a new call in the context of class instantiation. Those chunks are located in a certain memory region. Besides allocated chunks that represent in essence chunks currently in use, there are also *Freed Chunks* that represent chunks which were in use but no longer are. When a chunk gets freed, the chunk itself, or at least its data, stays in most scenarios at the same location as before, and its data is (beside some modifications which are explained later on) not deleted or overwritten.

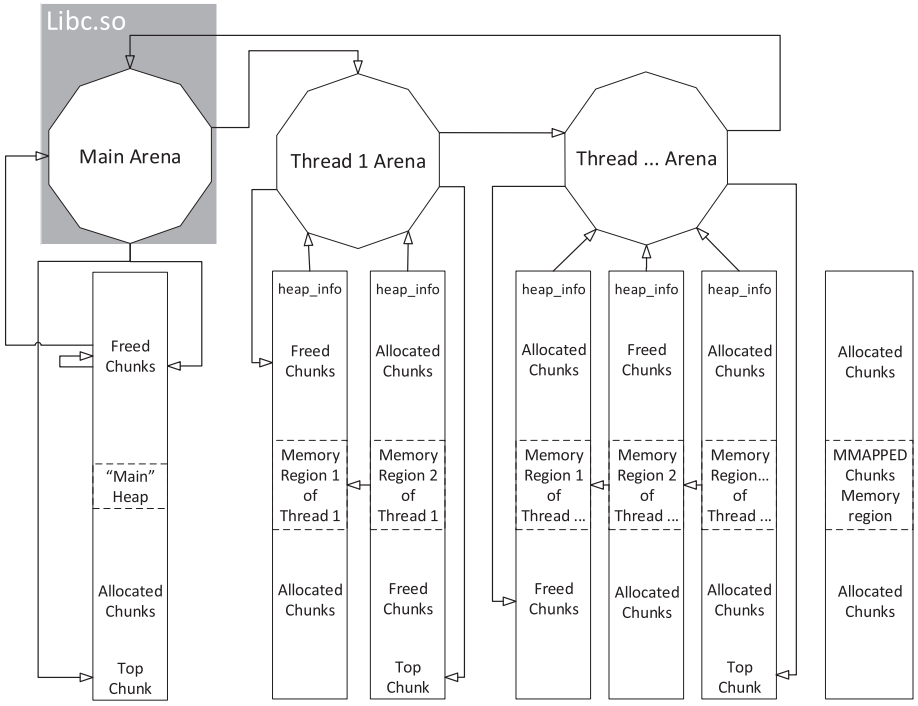

Fig. 1. Glibc heap overview.

At the highest level are arenas, which are described by a *malloc_state* struct. An arena is in essence heap space belonging to one or multiple threads while each arena has its own memory regions containing allocated and freed chunks from the associated thread(s). The arena contained in the Glibc library is called *main arena* as it is used by the first/main thread. While an arena does not have a direct link to each memory region or to allocated chunks (see Fig. 1), there are other connections like pointers to freed chunks, the next arena and the *top chunk*. This *top chunk* represents the remaining free space for a given arena, is used for the creation of new chunks and located at the end of an arena.

One level beneath arenas are the *heap_info* structs. Despite their name they do not describe the whole heap of a process or the part of the heap associated with a thread but only a part of a mapped memory region (described by an *vm_area_struct* struct) they belong to. More specifically, each mapped memory region belonging to an arena (except for the main arena) contains, at least at the beginning of the memory region, one instance of the *heap_info* struct, which holds the size of the current heap part in that memory region. Besides the size, each *heap_info* struct holds a pointer to the associated arena (*malloc_state* struct) and a pointer to the previous *heap_info* struct within the same arena (see also Fig. 3). In that way, all of them are linked together. The arena pointer is stored in the *ar_ptr* member and the reference to the previous *heap_info* in the *prev* member. In contrast to arenas, the *heap_info* structs are not linked circularly but instead, the *prev* field of the first *heap_info* is simply null.

Excluded from arenas and *heap_info* regions are MMAPPED chunks. As can be seen in Fig. 1, there are no links from MMAPPED chunks to any other structures or from heap structures to them. Those chunks are normally created when an allocation request exceeds a given threshold (typically 128∗1024 bytes). In that case, the chunk is not included in the main heap or any memory region belonging to another arena, but the operating system is asked for an exclusive memory region just for that chunk (via the *mmap* API call), in which the chunk is placed. As the *mmap* API call returns memory space in terms of pages, the minimum size of a *MMAPPED* chunk is one page (which is at least 4096 bytes or a multiple of that) and is evenly divisible by one page size. When an MMAPPED chunk is freed, the whole memory space it is allocating is removed from the process space and returned to the operating system.

The memory view

This section describes how and where the structs named in the previous section are stored in memory for a running Linux user space process.

Chunks in memory

The struct for a chunk is called *malloc_chunk* and contains the following fields:

prev_size If the previous chunk is freed, it contains the previous chunk's size.

size The distance from the beginning of the current chunk until the next chunk in bytes.

fd Forward link, pointing to the next freed chunk.

bk Backward link, pointing to the previous freed chunk.

fd_nextsize Points to the next chunk with a bigger size.

bk_nextsize Points to the next chunk with a smaller size.

This struct is located at the beginning of each chunk but not all fields are used for every chunk. In the case of allocated chunks, primarily the *size* field is used for its intended purpose. Only if the previous chunk is a freed chunk, is the *prev_size* field used. As already explained by Ferguson (2007, Section 0.2.1), the *size* field not only contains size information but the lower 3 bits of the size are used as special flags. The lowest bit (*PREV_INUSE* flag) indicates whether or not the previous chunk is a freed chunk, the second lowest bit (*IS_MMAPPED* flag) is set for MMAPPED chunks and the third lowest bit (*NON_MAIN_ARENA* flag) is set for chunks that are not part of the main arena (but not for MMAPPED chunks).

All other fields of the *malloc_chunk* struct are used for user/process data and do not contain any meta information (a technique called *boundary tags* Wilson et al., 1995, p. 28). This is illustrated in

Fig. 2, showing that user data of an allocated chunk (in this case the chunk at the top) reaches from the *fd* member until the beginning of the *size* field of the next chunk while filling all grey fields (marked as *Data*) with user data (including the next chunk's *prev_size* field).

The size and address of each chunk is aligned, which means it is evenly divisible by 8 on an ×86 and 16 on an ×64 architecture, respectively. In the case of MMAPPED chunks, the size and address is moreover evenly divisible by the page size. This is due to the *mmap* API function that returns the memory space for the requested chunk, which not only returns a size but also an address on a page size boundary. Moreover, despite the fact that each MMAPPED chunk results from a separate *mmap* call, multiple MMAPPED chunks can end up in the same memory region described by one *vm_area_struct* struct, as the kernel can simply enlarge a region. On the other hand, a continuous memory region can get split up in two separate regions if an MMAPPED chunk, located between two or more MMAPPED chunks from the same region, is freed (its related pages are returned to the operating system). Similar to *normal* allocated chunks, the user/process data starts right after the *size* field but does not include the next chunk's *prev_size* field because it is not guaranteed that an MMAPPED chunk is followed by another chunk.

As mentioned earlier, freed chunks are linked together and in that way form a bin. A bin can be seen as a container for freed chunks that always belongs to a specific arena. There are different types of bins, which mainly differ in the size of chunks they maintain. The types are *fastbins* (typically chunks of a size not larger than 80 bytes on ×86 architectures), small bins (typically not larger than 512 bytes on ×86 architectures) and large bins (everything above small bins). With the type, also the way of linkage changes and affects which fields of the *malloc_chunk* struct are used and hence how many bytes of user/process data gets overwritten on a free. Fastbin chunks only use the *fd* field, small and large bin chunks the *fd* and *bk* fields (building a circularly linked list), while large bin chunks use also the *fd_nextsize* and *bk_nextsize* fields. In each case, the according user data gets overwritten by the pointer(s). This is also true for the *prev_size* field, which gets set when the freed chunk is put in a small or large bin. So for example for a freed large bin chunk, the not overwritten user data reaches from after the *bk_nextsize* field until the beginning of the next chunk's *prev_size* field. The only exception regarding the *prev_size* field are fastbin chunks. For those chunks, all flags are unaffected, which means that e.g. the *NON_MAIN_ARENA* flag keeps being set but especially that the *PREV_INUSE* flag of the next chunk is not set on a free, resulting in not overwritten user/process data in the *prev_size* field.

The last case is the top chunk. As this chunk does not use any pointers, the data part starts like with allocated chunks right after the *fd* member and as it is the last chunk in an arena (and most of the time ends right at the memory region boundary), there is no following chunk whose *prev_size* field could be used.

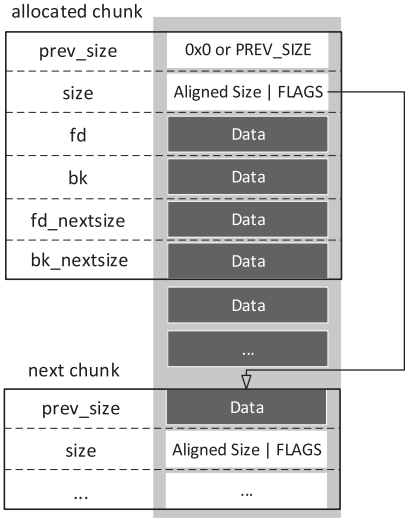

Fig. 2. Allocated chunk in memory.

Arena and heap info structs in memory

The main heap is a continuous region of memory containing all chunks of the main arena. While it is continuous, it can however get split up in multiple contiguous memory regions. Its describing *malloc_state* struct is stored in the *bss* section of the mapped Glibc library and as already stated, no *heap_info* structs are used for the main arena.

The *malloc_state* struct for thread arenas however is stored together with chunks in the same memory region. As can be seen in Fig. 3, it is located right after the first *heap_info* struct and before the first chunk of that arena. Normally, a *heap_info* stuct can be found at the beginning of each mapped memory region belonging to a thread arena. Besides that, it is possible that multiple *heap_info* stucts end up in the same mapped memory region. These *heap_info* stucts must not necessarily all belong to the same arena, but can also be related to different arenas. This is for example illustrated with the middle memory region in Fig. 3, where the lower *heap_info* struct belongs to another arena and the upper *heap_info* struct to the depicted *malloc_state* struct.

Heap and stack are normally at opposite sides and grow towards each other. This is, in fact, true as long as the heap does only consist of the main arena without any thread arenas or MMAPPED chunks. But as soon as either one of them is introduced, this strict separation is broken. Similar to MMAPPED chunks, the memory regions belonging to thread arenas most of the time get mixed up with memory regions containing stack frames for certain threads.

Plugin implementation

The Rekall Memory Forensic Framework (Google Inc, 2016c) offers a collection of plugins for the analysis of memory dumps. It originally has been forked from the Volatility (The Volatility Foundation, 2016) codebase back in 2013 and since then largely rewritten and extended. Our implementations have been tested with Rekall versions 1.5.1 and 1.5.2.post1. At the point of writing, they support at least the Glibc versions 2.20, 2.21, 2.22, 2.23 and 2.24 on an ×86 and ×64 architecture. The core component hereby is the Python class *HeapAnalysis*, which represents the implementation of all analysis results (partly) described before.

Based on the information explained in the previous sections, four plugins have been developed:

heapinfo Provides an abstract overview over the number of arenas, chunks and their sizes.

heapdump Dumps all allocated and freed chunks to disk in separate files for further analysis.

heapsearch Searches all chunks for the given string, regex or pointer(s).

heaprefs Examines the data part of the given chunk(s) for any references to other chunks.

While these plugins work most reliably with debug information for the Glibc version in use, the *HeapAnalysis* class provides functionality to analyze processes when no debug symbols are available.

The plugins will be demonstrated in Section Application on real world scenarios.

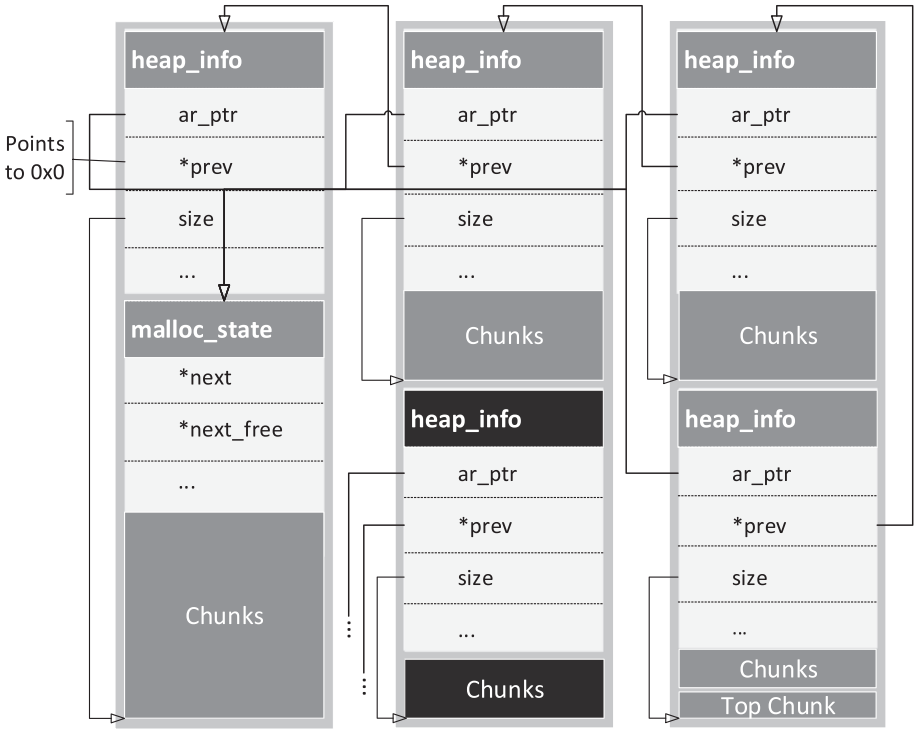

Fig. 3. malloc_state and heap_info structs in memory.

Getting the main arena

One of the first tasks the *HeapAnalysis* class is performing during its initialization is to locate the main arena. Besides holding important information such as the arena's size and the bin pointers, its location within the mapped Glibc library is a good indicator that the information examined during the search process are correct (the details will be explained in this section).

The most reliable method to get the main arena can be used if debug information, or more precisely, the constant offset for the *main_arena* symbol is available. If however this is not the case, the main arena cannot be directly determined and hence is searched via two different techniques. The first assumes that there is more than one thread and hence also more than one arena. As only the main arena struct is stored in the mapped Glibc library, all others are typically located at the beginning of a memory region, right after a *heap_info* struct. So, for each memory region described by a *vm_area_struct* struct (with some exceptions like mapped file regions or stack frames), the beginning of it is treated as a *heap_info* struct. If its *ar_ptr* field points right after itself and its *prev* field is null (the first *heap_info* struct points to no other *heap_info* instance and if it sits right before the arena, it is the first instance), the arena pointed to by *ar_ptr* is temporarily saved. If at the end at least one arena could be identified, it is tested whether or not following their *next* member ends up somewhere in the memory region of the mapped Glibc library. If that is the case, the address pointing in the Glibc is treated as the main arena.

If no further arenas could be identified (that means, there is only the main heap and potentially some MMAPPED chunks), the second technique is used. It leverages the fact that each bin chunk holds a circular doubly linked list. The idea is to follow the *bk* pointer of a bin chunk, until it leads to the main arena. To get such a bin chunk, every chunk on the main heap is examined for its *PREV_INUSE* bit. If this flag is unset in any chunk, the previous chunk is most probably a freed bin chunk. The previous chunk's *bk* field is now followed, chunk after chunk, in order to eventually end up in the main arena. If at some point, the *bk* field points into the mapped Glibc library, the main arena is most probably found.

In this scenario, there is however still a problem. The *bk* field points to the middle of the arena and not to the beginning. While it would be possible to exactly determine the position within that main arena by examining the chunk's size (each bin typically contains only chunks of a given size (range)) this approach is not always perfectly reliable because of varying bin sizes (the mapping between sizes and bins can be changed with compile time options). The method used instead to get to the beginning of the main arena is a search for the top chunk pointer. There are two reasons to use this approach:

- The offset of the top chunk pointer in the *malloc_state* is fixed for a specific Glibc instance and hence reliable. So, if the virtual address of that field is found, the distance to the beginning of the arena can be easily calculated.
- There needs to be a reliable process to correlate an expected field with the current value. The *top* field serves such a correlation, as the chunk that it points to, should reach until the end of the memory region (the top chunk's offset plus its size).

The process to get to the *top* field is now to walk backwards from the current bin, treat each pointer sized value as a pointer and check if it points inside the main heap and meets the requirements of the top chunk. As soon as the *top* field is found, the main arena's address is calculated by subtracting the *top* field's offset within the *malloc_state* struct from the found *top* field's memory address.

MMAPPED regions

While the identification of memory areas belonging to an arena is in most scenarios pretty reliable, identifying regions containing MMAPPED chunks is not. The reasons are:

- There is a lack of distinctive structs or reliable pointers to them.

- Any unnamed mapped memory region might contain MMAPPED chunks and they can be located anywhere in the process space.

It seems like pointers to those chunks are at most saved in stack frames of functions working with them, but not in any relevant book keeping struct. So one way to identify them could be to interpret all pointer-sized bytes from all stack segments (from all threads) as pointers, follow them and verify the information residing at that position. This approach has however two problems:

1. It is pretty error prone (interpreting arbitrary data as pointers) and, depending on the stack size, pretty time consuming (each pointer must be read, followed, the data it points to initiated as a chunk object and its values tested).
2. It might miss some MMAPPED chunks. In the case, where an MMAPPED chunk pointer is not used anymore, its value might get overwritten by newer stack frames, but if the chunk has never been freed, it still exists in the memory space, serving potentially valuable information.

Because of the lack of alternatives, the current approach to decide, whether or not a certain memory region contains MMAPPED chunks, is to perform a plausibility check. Because a memory region containing solely MMAPPED chunks (differing examples are described later in this section) also begins with an MMAPPED chunk, the first bytes are treated as such, and tested for the following characteristics:

- The *prev_size* field must have a value of zero.
- The chunk's size (the value of the *size* field without any flags) must be at least as large as the page size.
- The chunk's size must be evenly divisible by the page size.
- The chunk's offset plus its size must not exceed the boundary of the containing memory region.
- The location of the chunk must be evenly divisible by the page size (primarily useful for subsequent and hidden MMAPPED chunks).
- The *PREV_INUSE* and *NON_MAIN_AREA* bits must be unset and the *IS_MMAPPED* bit set.

Only if all of those properties are given, the corresponding memory region is considered to contain MMAPPED chunks. Those checks are also done on any subsequent data of the same memory region, before they get included as chunks.

While MMAPPED chunks are normally within an exclusive mapped memory region, it can happen that those chunks are placed at the bottom of a mapped memory region, containing different data such as a stack segment (see Fig. 4). As starting the search for MMAPPED chunks in the stack scenario at the beginning of the memory region might lead to false positives (interpreting stack data as chunks), a certain method is used, which is called *EBP unrolling* within this work. The basic approach is to follow all saved EBP pointers, starting with the base pointer gathered from the *pt_regs* struct (used to save register values on context changes). As each EBP value points to the next saved EBP, just following those pointers leads to the first saved EBP, probably located near the beginning of the stack segment. This process is also illustrated in Fig. 4.

Once the offset of the first saved EBP is identified, the next step is to search backwards, from this point on, for the first MMAPPED chunk. As MMAPPED chunks are only located at addresses evenly divisible by 4096 (minimum page size), only such addresses are examined. In cases where the EBP value does not point to a saved EBP in the stack segment (because it is e.g. used for carrying different data) the method stays basically the same, except it does

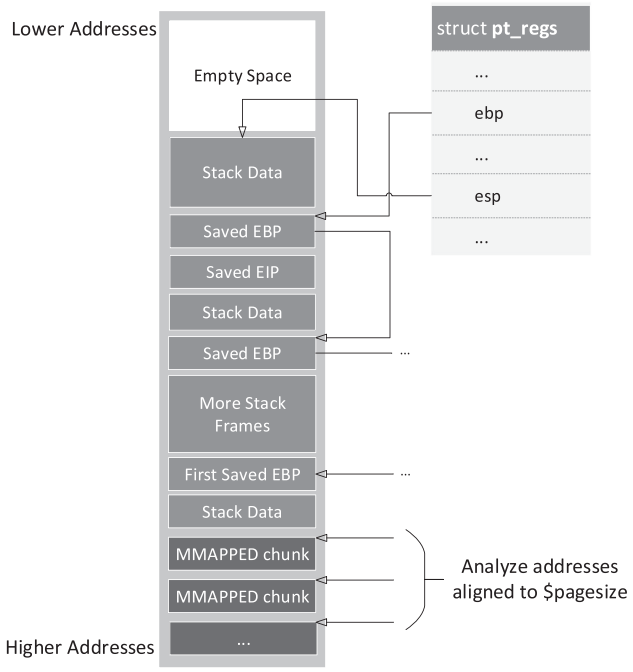

Fig. 4. Hidden MMAPPED chunks – EBP Unrolling.

no *EBP unrolling* and starts at the beginning of the memory region with the backwards search.

In cases where the missing MMAPPED chunks are not located after stack segments but *hide* somewhere else, the search scope must be extended while increasing the risk of false positives. The only regions that can be excluded are the stack and heap segments that have been already examined and those holding the content of mapped files (the *vm_area_struct* struct references a file object), as they could not be identified to ever contain other data (which would also be very unexpected, as those regions represent the file's content). The search process for this case stays the same, but without the *EBP unrolling*. After the first hit within a memory region (no matter if after a stack or somewhere else), this first MMAPPED chunk is being used to walk the potentially following chunks.

It should be noted that, in order to prevent false positives, the *HeapAnalysis* class starts a search for hidden MMAPPED chunks only if the current information about MMAPPED chunks seem to be incorrect, which will be explained in the following Section.

Evaluation

This section describes the evaluation of the *HeapAnalysis* class and its plugins.

Result verification

The verification of results is an important and elementary step in order to show the reliability of our methods and techniques. All tests have been conducted in the following environments while using different Glibc versions:

- Arch Linux 32 bit, ×86, Kernel Version 4.4.5-ARCH, Glibc Versions: 2.20, 2.21, 2.22, 2.23 and 2.24
- Arch Linux 64 bit, ×64, Kernel Version 4.4.5-ARCH, Glibc Versions: 2.20, 2.21, 2.22, 2.23 and 2.24

The *HeapAnalysis* class implements multiple functions, which compare the currently examined data with our expectations on

every chunk while processing memory:

- Test for correct flags (e.g. *NON_MAIN_ARENA* for chunks in a thread arena).
- Is the chunk's address aligned.
- Size checks (e.g. for alignment).
- Allocation status tests: Is a presumably allocated chunk part of any bin or fastbin?…
- In the case of MMAPPED chunks there are some additional tests (see Section MMAPPED regions for more details).

Regarding MMAPPED chunks, there is another verification step which involves the *malloc_par* struct. This struct's members are used by Glibc at the one hand to control certain global settings and on the other hand to count the total size and number of MMAPPED chunks. This is done via the only instance of that struct, called *mp_*, which is located in the mapped Glibc library. If the offset for *mp_* is provided, the *HeapAnalysis* class uses the struct to verify the number and size of all identified MMAPPED chunks. If there are any discrepancies, it tries to identify hidden MMAPPED chunks and if that does not resolve the issue, it prints a warning. Furthermore, the size of all chunks and heap related structs are compared with the sizes of the corresponding *vm_area* structs and the size of all arenas, respectively.

Completeness

While the results in this work do not cover 100% of Glibc's heap implementation, various steps have been performed to identify all relevant information and scenarios, relevant for the memory forensics perspective. We implemented a test set of programs that cover all special cases we are aware of to provide a ground truth and automatically verify the completeness of the output of our implementation using those test cases. Details about this set have been omitted here for the sake of conciseness, but are provided in our full technical report (Block and Dewald, 2017).

Application on real world scenarios

This section illustrates the application of our plugins on real world examples. This is on the one hand done by describing the analysis process itself and on the other hand by highlighting the advantage of using our plugins instead of a raw search through the entire heap. The analysis performed in the following subsections was done using a black box approach, which means that, if not specified otherwise, no process related details from the source code, like e.g. struct definitions, were necessary to be gathered beforehand.

zsh

The previously described *bash* plugin of Rekall and Volatility searches the whole heap space for timestamp strings that are prefixed with a hashtag, and afterwards searches it again for a history struct that points to the timestamp, in order to identify the issued command strings. The output of the plugin is a list of command entries, each consisting of the issued command and the corresponding timestamp. Our goal is to identify the same information for the zsh. The corresponding analysis in this section has been done in the same environments listed in Section Result verification.

The zsh process to analyze contained 142 executed commands in its history (when examining the history with the *history* command); the first one was `ps aux` and the last one `#$@%&*()`. The first part of the analysis process has been done in a black box approach, meaning no internal information about how zsh stores commands or time information have been gathered in any way beforehand. The only knowledge basis for this approach was the information already available regarding the bash command analysis. The first attempt was to find timestamp strings that are prefixed with a hashtag. Zsh does not however seem to store timestamps in the same way as bash. The next step was to search for issued commands somewhere in chunks using the heapsearch plugin. Listing 1 shows a result excerpt for searching the string `#$@%&*()` with heapsearch, revealing a chunk at address 0x09BCF830 which contains the command of interest.

```
Result for needle(s) #$@%&*()

[malloc_chunk malloc_chunk] @ 0x09BCF830
```

Listing 1. Using heapsearch to search for chunks containing issued zsh commands.

While commands could sometimes be identified in more than one chunk, each command seemed to be at least in one allocated chunk that only holds the command and some trailing bytes at the end. Listing 2 shows a hexdump of a chunk containing an issued zsh command.

```
2324 4025 262a 2829 0000 0000              #$@%&*().....
```

Listing 2. Hexdump of a chunk containing an issued zsh command.

As those chunks did not offer any meta information (e.g. at which time the command was issued), the next step was to find pointers to those chunks by again using the *heapsearch* plugin, but this time providing the address of the chunk that contains the issued command. It was possible to find exactly one pointer for each tested command in a separate allocated chunk with a size of 56 bytes (for the ×86 environment).

After examining multiple of those chunks (using the dumped content from the *heapdump* plugin), the following information could be derived:

- Bytes 5–8 contain a pointer to the issued command.
- Bytes 25–32 are 2 timestamps, stored as four byte integers. The first four bytes are the start time at which the command has been issued and the last four bytes are the time when the command ended.
- Bytes 41–44 contain the command counter.

When now examining those chunks for references using the *heaprefs* plugin (see Listing 3; the output has been stripped and modified for this work), it shows that bytes 1–4, 5–8 (the command pointer), 13–16, 17–20 and 33–36 point to other chunks. The start addresses of those chunks are listed in the *Comment* column, while the bytes containing the pointers are marked with square brackets in the *Data* column.

```
Examining chunk at offset 0x9C22938, belonging to the given
      address(es): 0x9c22938

                    Data                                  Comment
----------------------------------------  ------------------------
[b046c009] [38f8bc09] 02000000 [7081c209]   Chunk pointer: 9c046a8
                                            Chunk pointer: 9bcf830
                                            Chunk pointer: 9c28168
[c042c109] 00000000 65b27658 65b27658       Chunk pointer: 9c142b8
[a0d4c009] 01000000 e7060000 00000000       Chunk pointer: 9c0d498
 38000000  31000000 01000000 98510000
```

Listing 3. Analyzing a chunk for references with heapref.

By combining those insights with zsh's source code, the relevant history entry struct *histent* reveals two of those pointers: the fields *down* (bytes 17–20) and *up* (bytes 13–16). Those fields are used to reference the previous/next *histent* entry and hence allow a reliable traversal of *histent* instances. As the linked list of *histent* entries is circular, just walking one direction is sufficient to get all *histent* entries. The other pointers are not important for the current examination.

The last task at this point was to build a plugin, that automatically extracts those command information. To be able to traverse the *histent* list, the first step is to reliably identify one *histent* entry. As commands can be contained in chunks of various sizes and do not offer any searchable pattern, the approach is to find chunks containing the *histent* struct. The containing chunk's size is 56 bytes for ×86 and 96 bytes for ×64 architectures (the size results from the struct's size plus the bytes required to get an aligned chunk size; see Section Chunks in memory). Because there are also non relevant chunks with the same size, they need to be distinguished. As each *histent* entry should have a pointer to a chunk containing the command and a pointer to the next and previous *histent* entry, the test consists of checking whether or not those pointers reference an already known chunk. If the test result is positive, the last check is to walk the *up* and *down* pointers to the next and previous *histent* struct and test if their *down*/*up* member points to the current chunk. If this is the case too, the current chunk is treated as a *histent* struct and the command history is walked using the *down* member.

Listing 4 shows an example output of the zsh plugin (the output has been stripped).

```
PID   #          Started              Ended            Command
--- --- ------------------- ------------------- ------------
277   1  2016-03-31 23:51:09Z  2016-03-31 23:51:09Z  'ps aux'
...
277 142  2016-08-31 11:55:43Z  2016-08-31 11:55:43Z  '#!$@%&*()'
```

Listing 4. Example output for the zsh plugin.

While it was possible to reconstruct the *bash* history with a raw search (because of the timestamp string), this approach would not have worked for the *zsh*, as the timestamp is not saved as a string with an additional hashtag but only as a four byte integer. Because, in addition, no further searchable patterns could be identified during the analysis, a raw search is most probably not applicable in this context and hence shows the advantage of using the heap analysis plugins.

KeePassX

The second tool examined was the password manager *KeePassX* (version 0.4.3) and has been tested in the following environments:

- Ubuntu 15.10 32 bit, ×86, Kernel Version 4.2.0–16-generic, Glibc Version 2.21
- Ubuntu 15.10 64 bit, ×64, Kernel Version 4.2.0–16-generic, Glibc Version 2.21

The setup for the following analysis consisted of a *KeePassX* database, which contained several password entries that have been separated in two folders. Each password entry had a value for *Title*, *Username*, *URL* and *Comment*. When the database has been opened for analysis purposes, only the first folder was opened while leaving the second folder completely untouched.

Our first attempt was to find the unencrypted master password and the passwords of entries somewhere in the process space, but they could not be found. The unhidden password for a currently open password entry however, has been successfully observed in three allocated chunks during 5 tests with different password manager entries. The three chunks persist as long as the password entry shows the unhidden password. If the password is hidden again, two of the three chunks are freed but one stays allocated. Only if the entry window is closed, all chunks containing the password are freed. Depending on the size of the freed chunk, the password is overwritten within milliseconds up to a few minutes or even hours by a new allocation. While freed chunks, containing passwords of a length range from 1 to at least 40, are normally reallocated within a few seconds or minutes, there have been instances where a freed chunk containing a password of that size has been consolidated in a bigger freed chunk. As some bigger chunk sizes are not allocated that often (e.g. in the range of a few hundred bytes), the password might remain for probably a few hours in this chunk, but is also harder to find (it is surrounded by other data). Furthermore, the actual password has never been observed in the first 18 bytes of the chunks data part (see also the following analysis), which means that even in a case where the freed chunk is placed in a large bin, the password is not overwritten by any bin pointers on an ×86 architecture.

After the password field, the next step was to look for further fields of interest. The fields selected for this analysis were *Title*, *Username*, *URL* and *Comment*. *KeePassX* stores the full field content in allocated chunks right after the database has been opened. This is not only true for fields like the title, URL and comment, but also the username field, which in the case of *KeePassX* is shown only in asterisks in the overview and hence should not be needed unencrypted within the heap at that moment. To sum up: If a password database is opened and not locked, all fields from the overview (except the password field) from all password manager entries in all folders can be extracted from the heap. In order to analyze and compare the data from different chunks (containing the field strings), the *heapdump* plugin has been used to dump them in separate files. The following Listing 5 shows the hex dump output of a dumped chunk, containing a username (in this case yyyyyyyy_user5_AAAAAAAAB).

```
0100 0000 1900 0000 1800 0000 7258 4b09      ............rXK.
e0ff 7900 7900 7900 7900 7900 7900 7900      ..y.y.y.y.y.y.y.
7900 5f00 7500 7300 6500 7200 3500 5f00      y._.u.s.e.r.5._.
4100 4100 4100 4100 4100 4100 4100 4100      A.A.A.A.A.A.A.A.
4200 0000 0000 ffff ffff ffff                B...........
```

Listing 5. Hex dump of a chunk's data part, containing a KeePassX username field.

After comparing various chunks containing strings from the same type (such as usernames) and strings from other types, the following properties can be derived (which are also true for the unhidden password):

- The string is always 16-bit little endian encoded.
- The string does not start at the beginning of the chunk's data part, but exactly after 18 bytes. Most probably because the string is part of a struct/object.
- Bytes 5–8 and 9–12, respectively, correlate with the string's size, while bytes 9–12 state the correct size (the size is the number of characters represented by the encoded byte sequence, not the number of bytes). Both are probably instances of a four byte unsigned integer. The value from bytes 4–8 has

been exactly by one larger than the value from bytes 9–12 (see also Listing 5: 0x19 vs. 0x18).
- Bytes 13–16 point to the beginning of the string.
- The string is followed by 4 null bytes.
- Depending on the size of the string, there were additional bytes at the end (kind of padding bytes), ranging in the most cases from zero till 6 bytes. There have been however seldom cases, in which this number went up to 14 bytes (6 plus the amount of bytes until the next higher chunk size).

Bytes 1–4 did not change and while bytes 13–18 and the bytes after the string at the end changed a lot, they did not show any reliable coherence to a certain type or password manager entry.

The next step was to search for any pointers to a field string using the *heapsearch* plugin. While a search for the string's start address did not reveal any references (except for the one contained in the same chunk), searching for the beginning of the data part of that chunk revealed at least one pointer in another chunk. When analyzing this chunk with the *heaprefs* plugin, it reveals 12 pointers to other chunks. Following those pointers shows that four of them point to the chunks containing the *Title*, *Username*, *URL* and *Comment* strings. After analyzing more password entries, it was possible to make the fair assumption that for each password entry, there is a chunk of size 96 bytes (in the ×86 environment) that references at least those four fields. That means, by searching for chunks of the same size and examining the pointers at the given offsets, it is possible to gather the *Title*, *Username*, *URL* and *Comment* string of the same password entry. This information was used to create a proof of concept plugin by using the *HeapAnalysis* class, which automatically extracts these four fields for all password entries. Listing 6 shows an example output of that plugin (the output has been stripped, especially regarding the strings to fit in one line).

```
Entry      Title          URL        Username      Comment
------   -----------    --------    ---------    ------------
1        y_title1_A     y_url1_A    y_user1_A    y_comment1_A
2        y_title2_A     y_url2_A    y_user2_A    y_comment2_A
...
```

Listing 6. Example output for the KeePassX plugin.

It should be mentioned that without the information about the start address from the chunk context, finding references to the field strings would have been more difficult, while in the worst case preventing the possibility to correlate the various strings to one password entry.

Conclusion and future work

This section summarizes this work, highlights some limitations, gives a prospect on future work and concludes the results.

Summary

This paper focuses on analyzing the heap in the context of Linux processes with the research objective to support an investigator in analyzing data contained in user space processes. First, an in-depth understanding of Glibc's heap implementation was established, which is documented in Section Glibc analysis. The analysis focused on how and where heap related data is stored from a memory forensics perspective. Second, this knowledge has been used to build plugins for the memory forensics framework Rekall (Google Inc, 2016c), which analyze the heap of a Linux user space process and offer access to the identified chunks. The implementation details are documented in Section Plugin implementation and describe in particular an algorithm to identify hidden MMAPPED chunks. As producing reliable results is a crucial requirement in computer forensics, Section Evaluation covers information that enable the verification of the gathered results. To illustrate the usefulness of our implementation, Section Application on real world scenarios describes the black box analysis process of userspace applications, using the example of *zsh* and *KeePassX*.

Limitations

As already explained by Cohen (2015) for pagefiles, swap space is in some scenarios a crucial resource during the user space process analysis, which is not addressed in our work so far. When the plugins introduced in this work are used on a process with swapped out pages, containing heap related data, they will most probably fail in reliably analyzing the heap and extracting all chunks.

One of the goals of this work was to serve a process-like view on data contained in the memory. While this has been accomplished to the maximum extent of details that the heap is offering in this context (the location of a specific information and its size), it was not possible to extract information about the type of data. The reason is that the heap does not store any data type information. One way to still correlate a certain chunk with a specific type exists in a scenario where the data type and the size of the data itself is known up front. By searching for chunks of that size, the investigator might be able to gather data of a specific type. This approach requires however further tests on those chunks (as shown in Section zsh), as there might be more chunks with the same size containing different data.

As stated in Section Different heap implementations, the usage of a certain heap implementation is not bound to a specific operating system. In cases where a different heap implementation is used, the findings and plugins from this work will most probably not be applicable, but instead need to be performed analogue for those implementations. To this date, our *HeapAnalysis* class and the plugins only support the analysis of Linux processes on the mentioned architectures and Glibc versions. The information provided in this work and the technical report (Block and Dewald, 2017) can however be used to add support for further architectures or operating systems.

While it is possible to analyze the heap of a user space process without supplying debug information, it is not reliable in all cases. Beside the fact that the plugins will most certainly fail in analyzing the heap when any of the relevant structs has changed (*malloc_chunk*, *malloc_state* or *heap_info*), the results might still be incomplete if the pointer to the global variable *mp_* is missing. Without *mp_* it is not possible to reliably determine if all MMAPPED chunks have been discovered and as the search for hidden MMAPPED chunks is not initiated in this case, there might be at least one MMAPPED chunk missing in the output.

Besides hidden MMAPPED chunks, there is one further scenario, in which the *HeapAnalysis* class might miss chunks: If the analysis process mistakenly left out a *vm_area_struct* containing a whole arena, the results seem still to be correct but miss the arena's chunks. This case might happen in a scenario, where no debug information for the main arena's location have been provided and the main arena search process is not able to find it, which also means that this process was unable to find any thread arena. As no valid pointer to any arena is available in that scenario, there might be an undetected arena and hence unnoticed chunks. While this case is theoretically possible, the implemented functionality to detect such scenarios did not miss any arena during our evaluation.

Future work

The limitation considered most important is the missing support for swap space. In order to fully analyze the heap of a process,

the access to all pages containing heap related data must be ensured. The acquisition and integration of swap space into the memory forensics process is hence considered a fundamental step for further user space process analysis.

To increase the support of the *HeapAnalysis* class, the plugins could be tested on further architectures and adjusted appropriately if required. An analysis of further heap implementations such as *jemalloc* would in addition, allow to analyze the heap allocations of processes from applications such as Firefox. Furthermore, when conducting an appropriate analysis in the context of FreeBSD processes, it would also include another operating system.

Conclusion

The plugins introduced in this paper simplify the analysis process and enable the identification of information in memory that cannot easily be found using a pattern matching approach. These plugins and the documented details about heap objects in memory can also be used to support further research in the field of memory forensics and help forensic investigators to clarify an incident or crime. Furthermore, this paper demonstrated the analysis of user space processes while illustrating the advantage of having heap details during the analysis process.

References

Block, F., Dewald, A., 2017. Linux Memory Forensics: Dissecting the User Space Process Heap. Tech. Rep. Technical reports/Department Informatik – CS-2017-02. University of Erlangen-Nuremberg, Technische Fakultät [Visited on 12.04.2017]. https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/8340.

Case, A., Marziale, L., Neckar, C., Richard, G.G., 2010. Treasure and tragedy in kmem_cache mining for live forensics investigation. Digit. Investig. 7, S41–S47.

Cohen, M., 2015. Forensic analysis of windows user space applications through heap allocations. In: 3rd IEEE International Workshop on Security and Forensics in Communication Systems 2015, pp. 1138–1145 [Visited on 04.03.2016]. http://www.rekall-forensic.com/docs/References/Papers/p1138-cohen.pdf.

Ferguson, J.N., 2007. Understanding the Heap by Breaking It. Black Hat USA [Visited on 22.03.2016]. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf.

Free Software Foundation Inc, 2016. GNU C Library (glibc) [Visited on 15.08.2016]. https://www.gnu.org/software/libc/.

Gloger, W., 2006. ptmalloc2 [Visited on 15.08.2016]. http://www.malloc.de/en/.

Google Inc, 2016a. bash: Rekall Plugin to Scan the bash Process for History [Visited on 04.03.2016]. http://www.rekall-forensic.com/docs/Manual/Plugins/Linux/#bash.

Google Inc, 2016b. cmdscan: Rekall Plugin to Extract the Command History [Visited on 04.03.2016]. http://www.rekall-forensic.com/docs/Manual/Plugins/Windows/#cmdscan.

Google Inc, 2016c. Rekall Memory Forensic Framework [Visited on 04.03.2016]. http://www.rekall-forensic.com.

Leppert, S., 2012. Android Memory Dump Analysis. Master's thesis [Visited on 04.03.2016]. https://www1.informatik.uni-erlangen.de/filepool/thesis/android_memory_forensics.pdf.

Ligh, M.H., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons.

Macht, H., 2013. Live Memory Forensics on Android with Volatility. Master's thesis [Visited on 04.03.2016]. https://www1.informatik.uni-erlangen.de/filepool/publications/Live_Memory_Forensics_on_Android_with_Volatility.pdf.

The Volatility Foundation, 2016. Volatility [Visited on 04.03.2016]. http://www.volatilityfoundation.org.

Urrea, J.M., 2006. An Analysis of Linux RAM Forensics. Master's thesis. Naval Postgraduate School, Monterey, California [Visited on 04.03.2016]. http://calhoun.nps.edu/bitstream/handle/10945/2933/06Mar_Urrea.pdf.

Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D., 1995. Dynamic storage allocation: a survey and critical review. In: Memory Management. Springer, pp. 1–116.