Consistency Issue on Live Systems Forensics

Frank Y.W. Law, K.P. Chow, Michael Y.K. Kwan, Pierre K.Y. Lai

The University of Hong Kong
{ ywlaw,chow,ykkwan,kylai}@cs.hku.hk

Abstract

Volatile data, being vital to digital investigation, have become part of the standard items targeted in the course of live response to a computer system. In traditional computer forensics where investigation is carried out on a dead system (e.g. hard disk), data integrity is the first and foremost issue for digital evidence validity in court. In the context of live system forensics, volatile data are acquired from a running system. Due to the ever-changing and volatile nature, it is impossible to verify the integrity of volatile data. Let alone the integrity issue, a more critical problem – data consistency, is present at the data collected on a live system. In this paper, we address and study the consistency issue on live systems forensics. By examining the memory data on a Unix system, we outline a model to distinguish integral data from inconsistent data in a memory dump.

1. Introduction

Traditional computer forensics focuses on the examination of non-volatile data that are stored on digital storage media (e.g. hard disk) on an inactive system. These data exist permanently at a specific location under a defined format specified by the file system. Due to the static and persistent nature, its integrity can be verified in the course of legal proceedings starting from the point of acquisition to its appearance in court.

On a live system, some digital evidence exists in the form of volatile data, which is managed by the operating system in a dynamic environment. One example of volatile data is system memory data which contain information of processes, network connections and temporary data that are used by the operating system at a particular point of time. Unlike non-volatile data, memory data vanish and leave behind no trail after powering off the machine. There is no way to obtain the original content to verify the digital evidence obtained from the live system or the dump. For its high volatility and dynamicity, it is generally agreed that verifying the integrity of volatile data is impossible [20].

Memory data have become increasingly substantial at court proceedings. In particular, it is considered as a form of "electronically stored information" in a recent US judgment [1]. In order to produce digital evidence on a live system in court, it is essential to justify the validity of the acquired memory data. Recently, live system forensics has drawn remarkable attention among the research community of computer forensics. A number of research works [11, 12, 13] have been developed for conducting forensic investigation against a live system. To examine volatile memory data, one common approach is to acquire it into a dump file for offline examination. Various techniques [14, 15] have been proposed to shed light on the extraction of memory data from a live system. However, data consistency in connection with the dumped memory is rarely addressed. The problem of data consistency is described as follows: If a system is running, it is impossible to freeze the machine states in the course of data acquisition. Even the most efficient method would introduce a time difference between the moments that the first bit and the last bit are acquired. For example, the program may execute function A at the beginning of the memory dump and execute function B at the end of the dump. The data in the dump may correspond to different execution steps somewhere between function A and function B. As the data are not acquired at a unified moment, data inconsistency is inevitable on the memory dump.

Despite the acquired memory is inconsistent by itself, a considerable portion may constitute useful digital evidence. To moderate the contention and disputes in presenting this evidence before court, we study the data consistency problem with the view of proposing a model to distinguish data from inconsistent data in a memory dump.

The rest of this paper is organized as follows. Section 2 presents some common features and problems of the tools or techniques for acquiring volatile memory data. Section 3 elaborates the data consistency problem with respect to the dynamic memory structure. Section 4 discusses an experiment showing how to locate different segments of a running process in the memory. To conclude the paper, we suggest some future research directions in Section 5.

2. Literature Review

The concept of collecting volatile memory data is still new to computer forensic study and evolving researches have been given to this area with a view to search for proper ways of investigation into this area. Recently, the analysis of volatile memory data becomes an item in live incident response and there are a number of response toolkits being developed to address the needs [2, 3]. The available toolkits are often automated programs that run on the live system to collect transient data in the memory. However, if the response tool is run on a compromised system, the tool would heavily rely on the underlying operating system and may affect the reliability of the collected data [4]. Some of the response tools may even substantially alter the digital environment of the original system and causes an adverse impact to the dumped memory data. As a result, it is often required to study those changes to determine if those alterations will affect the acquired data [5].

Carrier and Grand [4] pointed out the potential flaws in acquiring volatile data through application running at the original system and proposed a hardware-based procedure for making a copy of memory contents to avoid the collected data being compromised by any untrusted code of the operating system or its applications. Antonio [6] further discussed the problems when acquiring live data through a network-based model and suggested a forensically sound approach in using firewire device to acquire memory data utilizing the Direct Memory Access (DMA) controller.

Notwithstanding, the aforementioned papers focus on methods and techniques that could be used for collecting reliable memory data, there are fewer analysis on the acquired memory data which contained transient and discrepant data. The inconsistency of memory violates computer forensic principles [16, 17, 18, 19] because data in the memory are not consistently maintained during system operation. This issue poses challenge for computer forensics and need to be addressed before presenting the evidence to the court of laws.

3. Consistency of Memory data

To study the consistent problem, we need to understand the basic structure of memory when a program is running. When a program is loaded into the memory, it will basically be divided into four segments, namely Code (*C*), Data (*D*), Heap (*H*) and Stack (*S*). The code segment contains the complied codes and all functions of a program, this is the area in which the executable instructions of the program reside. Normally, the data being stored in this segment are static and should not be affected whilst the memory is running. The other three segments contain data being used and manipulated by the program in running.

The data segment is used for global variables and static variables. The data in the data segment is quite stable and remains in existence for the duration of the process.

The heap segment is where dynamic memory for the program comes from. When dynamic memory is requested by the process using memory allocator such as *malloc*, the address space of the process grows upward. The data in the heap area can exist between function calls. They are less stable than the data in the data segment because the memory allocator may reuse memory that has released by the process.

The stack segment is where memory is allocated to local variables and parameters within each function and its data structure based on Last In First Out (LIFO) principle. When the program is running, the memory allocated to the stack area will be used by program variables again and again. This segment is the most dynamic area of the memory process and the data within the segment are discrepant and influence by various function calls of the program.

When the program is running, the code, data and heap segments are usually placed into a single contiguous area, whilst the stack segment is separated from those two segments and grows downward within the memory allocated space. The relationship among code, data, heap and stack segments is illustrated in Figure 1.

Indeed, the memory comprises a number of processes of the operating system or the applications running on top of it. It can be viewed as a large array that contains many segments for respective memory processes. In a live system, process memory "grows and shrinks" according to the system usage as well as user activities. As we can see from the above, the "grows and shrinks" of memory is either related to the growth of heap data or the expansion/release of stack data, whilst the data in code segment should be static and remain intact at all time. Apparently, data in the growing heap segment and stack segment cause inconsistency to the data that are contained in the memory as a whole, and the stack data affect more due to its vivid nature. Nevertheless, consistent data can be found at the code segment. By studying the behavior of data being contained in code, data, stack and heap segments, we may draw inference on the characteristics of consistent data that are existed at the memory being used by a specific operating system or application. These consistent data are usually dormant in nature and would not be affected when the process is running in the memory.

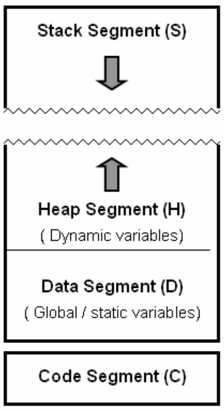


Fig. 1. Code, data, heap and stack segments in memory of a running process.

When obtaining a process dump of a running program, data in *C* remain unchanged because the memory segment only contain codes and functions of the program. Let's consider the global and static data, heap data and stack data which can be partitioned within a running process such that D_i, H_i, S_i (where *i* =1, 2, 3…) exist in different segments. In the ideal case, a snapshot is taken at a process at time *t*, the memory dump is consistent with respect to that point of time. The dump obtained would look like the one in Figure 2a.

In reality, however, it is impossible to obtain a memory snapshot as memory dumping takes time. The dumping process would span over a time interval δ. Should we obtain the dump at *t*, the actual dump obtained would look like Figure 2b, where x_i satisfies the following equation:

$$t \leq x_i \leq t+\delta \qquad (1)$$

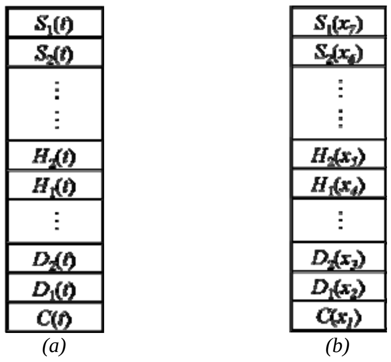


Fig. 2. *(a)* Memory dump obtained in ideal case. *(b)* Memory dump obtained in real case where x_i takes any value between *t* and $t+\delta$.

For a very fast memory dump, we should be able to have consistent data in *C*, whilst data in *D* may be consistent yet depends on the program usage. For the data in *H*, they are likely to be inconsistent because the segment may grow upon a call by the program. The data in *S* are highly likely to be changing due to its dynamic data structure, variables for the process functions are continually used and reused in this segment. This causes inconsistency to part of the dumped data collected.

In summary, it is considered the data exist in *C* is relatively static and their consistency should be preserved in the context of dumping the memory. The global and static data in *D* are less volatile, we may have a consistent view of them. Due to the special functionality of heap and stack, the data that are stored within those segments are more dynamic in nature and their consistency may give rise to new challenges under today's computer forensic standard.

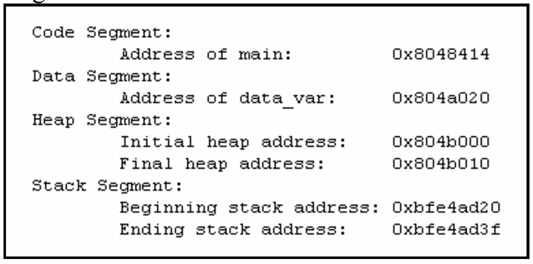
4. How to locate different segments in Unix

Each memory process has its own allocated space at the system memory which may include both physical and virtual addresses. To locate the heap segment, we can use the system call *sbrk* [8] to reveal the address space of a process. When *sbrk* is invoked with zero as the parameter, it returns a pointer to the current end of the heap segment.

We can also use the system command *pmap* [9] to display information about the address space of a process as well as the size and address of heap and/or stack segment. By using this information, we could eliminate the segment of heap/stack within a logical process dump [10] to achieve maximum data validity.

To identify the relevant parts of a memory process, a small program [21] is written to identify the Code, Data, Heap and Stack segments for the function *main()*. The corresponding code snippet is reproduced in

Appendix. The result of running the program is shown in Figure 3.



```
Code Segment:
        Address of main:          0x8048414
Data Segment:
        Address of data_var:      0x804a020
Heap Segment:
        Initial heap address:     0x804b000
        Final heap address:       0x804b010
Stack Segment:
        Beginning stack address: 0xbfe4ad20
        Ending stack address:     0xbfe4ad3f
```

Fig. 3. Locating various segments in a process

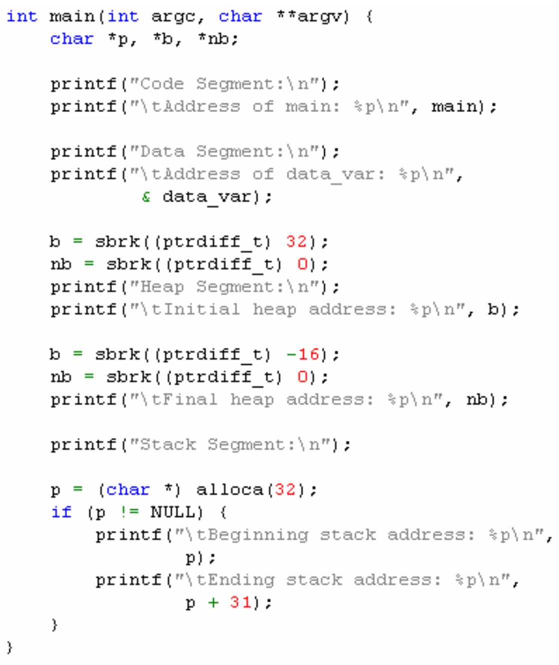
One can see the code segment for *main()*starts at the address `0x8048414`, which is followed by the data segment at the address `0x804a020`. The heap segment starts at the address `0x804b000`, which is immediately above the data segment. The stack segment grows downward from the initial address `0xbfe4ad20` to `0xbfe4ad3f`.

5. Conclusion and future works

It is no doubt that we require reliable tool and proper procedures to acquire memory data from live system to minimize any possible contamination to the collected data. Notwithstanding, due to the inconsistent nature of memory, the acquired memory data may raise challenge on its validity in the context of court proceedings. To overcome the problem, we discuss the component of memory and recommend the way of identifying consistent data that are contained within a memory process, such data are static in nature with its consistency is well-maintained. However, this research is only done on analyzing consistent data within a logical memory process, more work should be conducted to derive a method of identifying same kind of consistent data within the whole memory.

Volatile memory and live data collection are still green to the field of computer forensics and a substantial amount of researches still need to be conducted to secure the validity of the digital evidence collected from a live system.

6. Appendix - Code snippet of *main()*



```
int main(int argc, char **argv) {
    char *p, *b, *nb;

    printf("Code Segment:\n");
    printf("\tAddress of main: %p\n", main);

    printf("Data Segment:\n");
    printf("\tAddress of data_var: %p\n",
            & data_var);

    b = sbrk((ptrdiff_t) 32);
    nb = sbrk((ptrdiff_t) 0);
    printf("Heap Segment:\n");
    printf("\tInitial heap address: %p\n", b);

    b = sbrk((ptrdiff_t) -16);
    nb = sbrk((ptrdiff_t) 0);
    printf("\tFinal heap address: %p\n", nb);

    printf("Stack Segment:\n");

    p = (char *) alloca(32);
    if (p != NULL) {
        printf("\tBeginning stack address: %p\n",
                p);
        printf("\tEnding stack address: %p\n",
                p + 31);
    }
}
```

7. References

[1] Judgment from Honorable Jacqueline Chooljian, *Verdict of United States District Court*, http://i.i.com.com/cnwk.1d/pdf/ne/2007/Torrentspy.pdf

[2] Harlan Carvey, *Windows Forensics and Incident Recovery*, Addison Wesley, 2005.

[3] Kevin Mandia, Chris Prosise, and Matt Pepe, *Incident Response and Computer Forensics*, McGraw-Hill Osborne Media, 2 edition, 2003.

[4] Brian D. Carrier and Joe Grand, A Hardware-Based Memory Aquisition Procedure for Digital Investigations, *Journal of Digital Investigations*, 901(1), 2004.

[5] A.Walters and N. Petroni, Jr, *Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process* Komoku, Inc., College Park, MD, USA, Jan 2006, http://www.komoku.com/forensics/asic/bh-fed-07-walters-paper.pdf

[6] A. Martin, *FireWire Memory Dump of Windows XP: A Forensic Approach*, Boston University, April 2007, http://www.friendsglobal.com/papers/FireWire%20Memory%20Dump%20of%20Windows%20XP.pdf

[7] III Golden G. Richard and Vassil Roussev, *Next-generation digital forensics*, Commun, ACM, 49(2):76–80, 2006.

[8] Unix Manual for command *sbrk*, http://www.scit.wlv.ac.uk/cgi-bin/mansec?2+sbrk

[9] Unix Manual for command *pmap*, http://www.scit.wlv.ac.uk/cgi-bin/mansec?1+pmap

[10] S.H. Lam, *Computer Forensic Analysis*, Chinese University of Hong Kong, October 2000, http://home.ie.cuhk.edu.hk/~shlam/ssem/for/

[11] F. Adelstein, Live forensics: Diagnosing your system without killing it first. Feb 2006/ Vol.49, *Communications of the ACM*, 49(2), 63-66.

[12] M. Burdach, *Forensic Analysis of a Live Linux System* (2 parts). http://www.securityfocus.com/infocus/1769, http://www.securityfocus.com/infocus/1773

[13] M. Burdach, *An introduction to Windows memory forensic*. http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf

[14] B. D. Carriera, J. Grand, A Hardware-Based Memory Acquisition Procedure for Digital Investigation, *Digital Investigation Journal* 1(1):50-60, ISSN 1742-2876, February 2004.

[15] G. M. Garner, *Forensic Acquisition Utilities*. http://users.erols.com/gmgarner/forensics/

[16] Information Security and Forensics Society (ISFS), *Computer Forensics, Part 2: Best Practices*, May 2004, http://www.isfs.org.hk/publications/ComputerForensics/ComputerForensics_part2.pdf

[17] Thomas Sudkamp, Inference propagation in emitter, system hierarchies, Proceedings of the ACM SIGART international symposium on Methodologies for intelligent systems, *International Symposium on Methodologies for Intelligent Systems*, 1986, ACM Press New York, NY, USA, pp 165-173

[18] B. Westbrook and B. Zornado, *Proposal for electronic records management task force*, 18 October 2001, http://ww.uclibraries.net/sopag/erm/ERMTFReport.pdf

[19] Amihai Motro, Philipp Anokhin and Aybar C. Acar, Utility-based Resolution of Data Inconsistencies Information Quality in Informational Systems, *Proceedings of the 2004 international workshop on Information quality in information systems*, ACM Press New York, NY, USA, pp 35 – 43.

[20] M. Burdach Finding Digital Evidence in Physical Memory, *BlackHat Federal*, January 2006. http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Burdach/bh-fed-06-burdach-up.pdf

[21] A. Robbins, *User-level Memory Management in Linux Programming*, 13 May 2004, http://www.phptr.com/articles/article.asp?p=173438&seqNum=2&rl=1