Utilizing Program’s Execution Data for Digital Forensics

Ziad A. Al-Sharif
Software Engineering Department
Jordan University of Science and Technology
Irbid, 22110, P.O. Box 3030, Jordan
zasharif@just.edu.jo

ABSTRACT
Criminals use computers and software to perform their crimes or to cover their misconducts. Main memory or RAM encompasses vibrant information about a system including its active processes. Program’s variables data and value vary in their scope and duration in RAM. This paper exploits program’s execution state and its dataflow to obtain evidence of the software usage. It extracts information left by program execution in support for legal actions against perpetrators. Our investigation model assumes no information is provided by the operating system; only raw RAM dumps. Our methodology employs information from the target program source code. This paper targets C programs that are used on Unix based systems. Several experiments are designed to show that scope and storage information of various source code variables can be used to identify program’s activities. Results show that investigators have good chances locating various variables’ values even after the process is stopped.

KEYWORDS
Digital Forensics, Memory Forensics, Memory Dumps, Carving Variable Values, String Variables, C Programs.

1 INTRODUCTION
Criminals use computers and software to perform their crimes or to cover their wrongdoings. Locating a program on the machine’s hard disk might not be enough to establish the definite usage of that program. An evidence might be needed to confirm that the perpetrator is actually used that program. This evidence can be found in a couple of places, one of which is the RAM of the used machine. This emphasizes the significance of memory forensics and its use in crime investigation. Generally, programs vary on their dependency on memory, CPU, disk I/Os, and networks [1]. A program’s control flow might highly depend on various variables and their values that are stored in different main memory locations (RAM). These variables can be categorized based on their scopes and execution lifetimes. A scope determines the visibility of a variable and where it can be accessed within the program’s source code. In contrast, variable’s storage (memory type) determines the duration in which its value is created and destroyed or deleted. Additionally, variables can be classified based on whether they are allowed to be changed during execution. Constant variables are those that cannot be changed once are assigned, most of which are often assigned with literal values (hard coded values). These literals might be unique to the executable program and its execution state. Additionally, many other non-constant variables might be initialized with hard coded values (literals).

Assuming no information is available from the operating system; only raw RAM dumps. This paper locates evidences that would be used to confirm the software usage and its association with the crime. Our investigation model is based on variables’ scopes and memory types. In order to verify our research methodology, various experiments and scenarios are developed. RAM memory dumps are created and analyzed to locate related variables’ value (literal and non-literal) based on the program source code and its execution state.

This paper targets C programs that run under Unix based systems. However, most of our findings are equally applicable for other
languages and operating systems. Our results show that regardless of whether the process is active or just stopped, the memory investigator can employ knowledge about the program source code and its variables such as global and local static and their potential values to assure the program usage. Hence, values of local auto variables are successfully located when their corresponding stack frames are still active. On the other hand, dynamically allocated values can be located as long as the program is not stopped and the corresponding memory is not released.

The rest of this paper is organized as follows. Section 2 highlights some of the background knowledge used in this paper. Section 3 describes our investigation model and how it employs information available in the program source code to confirm that the program is actually used. Section 4 presents our four experiments and Section 5 discusses our promising results. Section 6 presents some of the related works. Finally, our planned future work is presented in Section 7 whereas Section 8 concludes our findings.

2 BACKGROUND

A software process may employ various variables (memory storages). In C language, variables can be classified based on their scope and duration into global, local auto, and local static [2]. Global and static data is allocated by the runtime system for a program at its start. These variables might be initialized with default values whenever they are not explicitly initialized by the programmer. The lifetime (duration) of this kind of data is same as the process that uses the data [3, 4]. However, unlike global variables, the visibility of local static variables is limited to the scope of its function or block.

Typically, all operating systems provide services to programs they run. In a Unix based system, when the Kernel executes a C program, a special routine (known as the startup routine) is automatically invoked to set up the command line arguments and the environments. Then, the main() function is called.

The Kernel manages software processes, each of which is provided a dedicated memory space in RAM [5, 6, 7]. When the executable starts, various sections are allocated and loaded into RAM, the starts and ends of these sections are independent of the RAM page limits. During execution, different variables are stored in RAM into various logically classified segments. Figure 1 shows a logical view of the major memory segments that are dedicated to a running process. A loaded process consists of the following major segments:

.text: is a read-only segment that contains the binary instructions (executable) located below the heap and stack.

.rodata: is a read-only segment that contains the immutable variables; read-only constants and string literals.

.data: is a read-write segment that contains global and local static mutable variables that are explicitly initialized by programmers.

.bss: is a read-write segment that contains global and local static mutable variables that
are uninitialized by the programmer. Usually, variables in this segment are initialized by the Kernel before the program starts.

**Heap:** is a memory segment allocated when the process starts. It provides runtime memory allocation for variables and their values as needed during execution. Program’s data that lives in heap can be referenced outside the function scope. In C language, the heap memory allocations are managed by the program with help from the Kernel through system calls such as `malloc()`, `calloc()` and `realloc()`. An explicit request can be initiated to release these allocated memory using system calls such as `free()` [3, 4].

**Stack:** is a memory segment allocated when the program starts and it is automatically managed by the Kernel and its runtime system. It consists of blocks called activation records or frames, each of which represents a call to a function and provides storage for its corresponding local and formal parameters. The lifetime (duration) of variables allocated on the stack is same as the scope in which they are declared (mostly the function and its stack frame) [3, 4].

Figure 2 shows a sample C program with various variables’ scopes and their correspondence to the logical view presented in Figure 1. Hence, memory investigators can utilize the memory of various variables’ values to locate evidences about the actual use of the software.

### 3 INVESTIGATION MODEL

A variable scope affects its visibility within the program source code and a variable storage affects its duration. Hence, different scopes and storages might affect the survivability of a variable’s value in memory during various execution states. Accordingly, locating these values in a RAM memory dump can be used as evidence to prove that a user is actually used the presumed program. Our investigation model studies the possibility of locating these variables’ values that are used within different scopes and storage types. Our experimentations study three different scopes: global, local auto, and local static. It also tries to distinguish between various values within different execution states, see Figure 3. These execution states are based on various scenarios such as:

- The variable is used or not-used yet during program execution
- The variable was used in a currently active or inactive stack frame
- The variable is never used; the variable is never reached or the stack frame is never been active
- The allocated variable’s data is never released or just released
- The software process is live (still running) or dead (just stopped)

Furthermore, our investigation model assumes no information is provided by the operating system, only memory dumps that are created during various execution states and scenarios.
Then, each memory dump is searched for potential values related to the source code of the presumed program, see Figure 4.

![Investigated Execution States & Variables’ Values](image)

**Figure 3.** Various variables’ states that are explored during our experiments. #1 represents a literal value within an active frame and a live process. #2 represents a literal value within a currently inactive frame and a live process. #3 represents a literal value within a currently inactive frame and a dead process. #4 represents a dynamically allocated value first within a live process and then within a dead process.

4 EXPERIMENTS

Four experiments are designed, each of which explores the potential evidence that would prove the actual software usage during various execution states, see Figure 3. A variable can be assigned a literal or non-literal value. Non-literal values are those that are dynamically calculated or modified and assigned during program execution. Thus, experiments 1, 2, and 3 are designed to investigate variables assigned with string literals during different execution states. On the contrary, experiment 4 is designed to investigate variables’ values that are dynamically allocated and modified. Our experiments explore three different variable’s scopes: *global*, *local auto*, and *local static*.

**Experimentation Setup:** In all four experiments, we used a Linux virtual machine that is created using VirtualBox. The VM runs openSUSE Linux version 13.1 with 512 MB of RAM memory. This VM is hosted on a Mac OS X 10.11.5. See Figure 5.

4.1 Experiment #1

First experiment is designed to explore the use of a literal string in a *currently active* stack frame and whether it affects the ability to locate this string in a memory dump that is created during a live process, see #1 in Figure 3. It investigates three different variable scopes: *global*, *local auto*, and *local static*, each of which is initialized at declaration time.

![Investigation Model](image)

**Figure 4.** Investigation Model: Step 1 represents the process of creating a memory dump. Step 2 represents the searching process for potential values related to the target program source code.
with a string literal. It explores two different states within an active stack frame of an active process:

- **State 1**: The variable is used; reached in one of the executed statements
- **State 2**: The variable is not-used yet; not reached in any of the thus far executed statements

A memory dump is seized in each of these states for each of the explored variables. Each of these dumps is searched for the subject variable and its literal string value. The results of our findings are presented in Section 5.

### 4.2 Experiment #2

Second experiment is designed to explore the use of a literal string in a currently inactive stack frame and whether it affects the ability to locate this string in a memory dump, see #2 in Figure 3. Similar to the first experiment, this one targets live processes with three different variable scopes: global, local auto, and local static, each of which is initialized at declaration time with a string literal. It explores three different variable’s states within an inactive stack frame of an active process:

- **State 1**: The variable was used; read or assigned in one of the executed statements
- **State 2**: The variable was not used; not read or assigned in any of the thus far executed statements
- **State 3**: The stack frame was never active; the function is never called

### 4.3 Experiment #3

Third experiment is very similar to the second experiment. Except, it explores the effects of having an inactive process (just stopped) on the same three scopes and the same three states investigated during the second experiment, see #3 in Figure 3. A memory dump is seized in each one of these states for each of the three explored variables’ scopes. Then, these dumps are searched for these literal values. Results of our findings are discussed in Section 5.

### 4.4 Experiment #4

Fourth experiment is designed to investigate the dynamically allocated string variables and contrast them with variables that are initialized with literal values in the source code. This experiment explores the possibility of locating variables’ values that are allocated in the heap memory, see #4 in Figure 3. In this experiment, variables are dynamically allocated using `malloc()` and assigned a string value from another string literal using the `strcpy()` function, then some characters are modified to distinguish the string resides in the dynamically allocated heap space from the original literal string. It explores the potential of locating these string values in four different states:

- **State 1**: `malloc()`, `strcpy()`, and one character is modified
- **State 2**: `malloc()`, `strcpy()`, one character is modified, then the `free()` function is called
- **State 3**: `malloc()`, `strcpy()`, one character is modified, the `free()` function is called, and then the process is terminated normally
States 4: `malloc()`, `strcpy()`, one character is modified, and then signal SIGINT (Control-C) is used to terminate the program abnormally. No `free()` is called explicitly by the user program.

States 3 and 4 are designed specifically to explore the consequences of having a process that is terminated normally and a process that is terminated abnormally. A memory dump is seized in each one of these states. Then, these memory dumps are searched for these dynamically allocated and modified string values. Results of our findings are discussed in Section 5.

5 RESULTS

This section thoroughly presents the results from all four experiments.

**Results from first experiment:** show that the values of global and local static variables have two occurrences each. Whereas, the value of local auto variable has only one occurrence in both states (State 1 and State 2), see Table 1. This means, in an active stack frame, having the referenced variable used or not-used does not affect the number of occurrences that can be found of the searched value. It also means that the investigator can find double occurrences of global and local static variables initialized with literal strings and only one occurrence can be found for local auto variables.

**Table 1.** Results from the first experiment show that global and local static variables have two occurrences whereas the local auto variable has only one occurrence for its value. States 1 & 2 show that having the variable used or not-used does not affect the number of occurrences in all investigated scopes.

<table>
<thead>
<tr>
<th>Var. Scope</th>
<th>State 1</th>
<th>State 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Local (auto)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Local Static</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Results from second experiment:** show that the values of global and local static variables are found twice in the RAM dump (two occurrences). However, the value of local auto variable is never found in any of the three investigated states. This means that having an inactive stack frame during a live process reduces our chances of locating the values of local auto variables to zero. Whereas, having an active or inactive stack frame does not affect the values of global and local static variables; at least in our investigation setup, which consists of relatively small programs. Table 2 presents our findings for each of the three different variables’ scopes and each of the three investigated states.

**Table 2.** Results from the second experiment show that global and local static variables have two occurrences whereas the local auto variable has zero occurrence for its value. States 1, 2, & 3 show that having the variable used or not-used does not affect the number of occurrences in all investigated states, even when the stack frame is never active.

<table>
<thead>
<tr>
<th>Var. Scope</th>
<th>State 1</th>
<th>State 2</th>
<th>State 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Local (auto)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Local Static</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Results from third experiment:** show that the value of the local auto variable is never found in any of the three investigated execution states (zero occurrence). This goes along with the results from second experiment. However, the number of occurrences of values of global and local static variables is decreased from two occurrences to only one occurrence for each variable in each state. This means, if the process is inactive (dead), the investigator has a one chance to locate literal values of global and local static variables (one occurrence). Table 3 presents our findings for each of the three different scopes and each of the three investigated states.

**Results from fourth experiment:** show that the investigator have a chance to locate a dynamically allocated string value that resides in the heap memory for a dynamically allocated variable as long as its process
Table 3. Results from the third experiment show that global and local static variables have only one occurrence whereas the local auto variable has zero occurrence for its value in the RAM dump (when the process is inactive). This is true in all of the three states, which means having the variable used or not use does not affect the results but having an active or inactive process does affect the results. States 1, 2, & 3 show that having the variable used or not-used does not affect the number of occurrences in all investigated states, even when the stack frame is never active.

<table>
<thead>
<tr>
<th>Var. Scope</th>
<th>State 1</th>
<th>State 2</th>
<th>State 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Local (auto)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Local Static</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

is live and the value is not released yet (*free()* function is not called) explicitly in the program. Otherwise, we have a zero chance of locating any of these string values; at least in our experimentation setup. Table 4 shows the number of occurrences for the investigated string value within four different execution states.

Table 4. Results from the fourth experiment show that global, local auto, and local static variables have only one occurrence in State 1 (where the process is active and the (*free()* function is not called). Whereas, all variables’ scopes have zero occurrences in all of the other three states. This means we have a chance to locate dynamically allocated strings only in State 1.

<table>
<thead>
<tr>
<th>Var. Scope</th>
<th>State 1</th>
<th>State 2</th>
<th>State 3</th>
<th>State 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Local (auto)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Local Static</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

6 RELATED WORK

Many researchers find in the RAM memory a vital source of information that can be used in support for legal actions against criminals in digital forensic cases [8, 9, 10, 11, 12, 13]. Ahmad Shosha et al. developed a prototype to detect different malicious programs that are regularly used by criminals. The proposed approach depends on the deduction of evidences that are extracted based on traces related to the suspect program [14]. Chan Ellick et al. introduced ForenScope [15] a RAM forensic tool that permits users to investigate a machine using regular *bash-shell*. It allows users to disable anti-forensic tools and search for potential evidences. In order to maintain the RAM memory intact, it is designed to work in the unused memory space on the target machine. Petroni et al. introduced FATKit [16]. It is a digital forensic tool dedicated to extract, analyze, and visualize the digital forensic data. It utilizes program source code and its data structure during the analysis of memory dumps. Arasteh et al. extracts evidences from RAM memory based on the logic of the process that is extracted from its stack memory segment [17]. Funminiyi Olajide et al. uses RAM dumps to extract user’s input information from Windows applications [18]. Narasimha Shashidhar et al. targeted the *prefetch* folder and its potential value to the investigator. This prefetch folder is used to speed up the startup time of a program on a Windows Machine [19].

7 FUTURE WORK

For future work, we are planning to investigate other environments: Windows, Mac, and small devices such as phones and tablets. Some languages have its own memory management system and its own virtual machine while the other just like C depends directly on the operating system in their memory management. We plan to investigate the differences in the behavior of various programing languages such as C++, Java, C#, and Python. Furthermore, we are looking forward to investigate similar scenarios for other data types and data structures. Finally, it would be important to investigate various types and their impacts on long running programs such as servers.

8 CONCLUSION

This paper utilizes information from the source code of the a program and employs program’s execution data during various execution states to help investigator establish the evidence against a perpetrator. This will allow law enforcements to take legal
actions against criminals in the court of law. Our experimentation is based on the C programming language. Based on these experiments, we found that utilizing source code information can be valuable to the investigator. It helps establish the evidence that the perpetrator is actually used the software to perform the crime or to cover the wrongdoing. Various string literals and non-literals related to the program execution are successfully located during various scenarios and execution states.

REFERENCES


