Forensic Memory Analysis: From Stack and Code to Execution History

By

Ali Reza Arasteh and Mourad Debbabi

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2007 USA

Pittsburgh, PA (Aug 13th - 15th)

available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

Forensic memory analysis: From stack and code to execution history☆

*Ali Reza Arasteh**, *Mourad Debbabi*

Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada

Keywords:
Cyber forensics
Physical memory
Stack
Thread
Process logic

ABSTRACT

Forensics memory analysis has recently gained great attention in cyber forensics community. However, most of the proposals have focused on the extraction of important kernel data structures such as executive objects from the memory. In this paper, we propose a formal approach to analyze the stack memory of process threads to discover a partial execution history of the process. Our approach uses a process logic to model the extracted properties from the stack and then verify these properties against models generated from the program assembly code. The main focus of the paper is on Windows thread stack analysis though the same idea is applicable to other operating systems.

1. Motivations and background

Nowadays, more and more cyber attacks are affecting corporate and government networks and sometimes even the IT systems underlying the critical infrastructure. These attacks raise major concerns from the law enforcement standpoint. Owing to the borderless nature of cyber attacks, many criminals/offenders have been able to walk away due to the lack of supporting evidence to convict them. In this context, cyber forensics plays a major role by providing scientifically proven methods to gather, process, interpret, and use digital evidence to bring a conclusive description of cyber crime activities. In the commercial software market flooded by security products, the development of forensics IT solutions for law enforcement has been limited. Though outstanding results have been achieved for forensically sound evidence gathering, little has been done on the analysis of the acquired evidence. This is particularly true for volatile evidence sources such as physical memory and cache which is mainly due to the volatile and unstable nature of data which is residing on these media. However, if not damaged, the information that is acquired from such sources is the most pertinent and definitive evidence which should be analyzed during the initial phases of investigation. In this paper, we propose a new technique in memory analysis and present the system that has been developed to realize the application of this technique. The proposed technique would help forensics analysts to determine what a process has done during the incident by recovering the chain of function calls it has made during its execution. The technique consists of the following phases:

- Parsing the internal memory structures.
- Retrieving the process assembly code and stack from the memory.
- Constructing the CFG from the executable code.
- Modeling the program execution by transforming the CFGs to local automata and combining the local automata models into a Push Down System (PDS).
- Modeling the stack residues properties using process logic.
- Extracting all the possible execution paths by correlating the stack residues properties and the program execution model.

☆ The research reported in this paper is the result of a fruitful collaboration with Bell Canada under the PROMPT Quebec research partnership program.

* *Corresponding author.*
E-mail addresses: a_araste@ciise.concordia.ca (A.R. Arasteh), debbabi@ciise.concordia.ca (M. Debbabi).

1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.010

In this paper, for the sake of illustration, we focus on Windows operating system. However, the same approach is applicable to any operating system that applies stacking mechanisms to handle function calls. The paper is organized as follows. In Section 2, the related work on forensic analysis and physical memory analysis, though limited, is discussed. In Section 3, a summary of our approach is presented and Section 4 details different aspects of our approach by introducing the problem, and then detailing the solution. Section 5 gives a general overview of the pertinent kernel data structures and Windows memory layout. Section 6 summarizes our discussion by providing the conclusion and future research directions.

2. Related work

In spite of the limited research result available on forensic analysis, there are some important proposals that we detail hereafter. The state of the art on cyber forensic analysis could be categorized as follows: baseline analysis, root cause analysis, common vulnerability analysis, timeline analysis, semantic integrity check analysis and memory analysis.

Baseline analysis, proposed in Monroe and Bailey (2003), uses an automated tool that checks for the differences between a baseline of the safe state of the system and the state during the incident. An approach to post-incident root cause analysis of digital incidents through the separation of the information systems into different security domains and modeling the transactions between these domains is proposed in Stephenson (2003).

The common vulnerability analysis (Tenable Network Security), involves searching through a database of common vulnerabilities and investigating the case according to the related past and known vulnerabilities. The timeline analysis approach (Hosmer, 1998) consists of analyzing logs, and scheduling information to develop a timeline of the events that led to the incident. The semantic integrity checking approach (Stallard and Levitt, 2003) uses a decision engine that is endowed with a tree to detect semantic incongruities. The decision tree reflects pre-determined invariant relationships between redundant digital objects.

Gladyshev and Patel (2004) propose a formalization of digital evidence and event reconstruction based on finite state machines. Other research on formalized forensic analysis includes the formalization of event time binding in digital investigation (Gladyshev and Patel, 2005; Leigland and Krings, 2004), which proposes an approach to constructing formalized forensic procedures. The absence of a satisfactory and general methodology for forensic log analysis has resulted in ad hoc analysis techniques such as log analysis (Peikari and Chuvakin, 2004) and operating system-specific analysis (Kruse and Heiser, 2002).

The DFRWS memory forensics challenge (DFRWS, 2005) is considered as one of the initiatives for the research on memory analysis. The challenge led to the development of two memory analysis tools: Memparser (Betz, 2005) and Kntlist (Garner) each capable of traversing the link list of process structures kept by the operating system to extract information about a running process. Burdach presents an approach to retrieve process and file information from the memory of Unix operating system by following the unbroken links between data structures in the memory. The work in Walters and Petroni presents an extensible framework (FATKit), which provides the analyst with the ability to automatically derive digital object definitions from C source code and extract the underlying objects from memory. Walters and Petroni (2007) present an approach for extracting in-memory cryptographic keying material from disk encryption applications.

Schuster (2006) proposes an approach to define signatures for executive object structures in the memory and recover the hidden and lost structures by scanning the memory looking for predefined signatures. However, defining a signature that uniquely identifies most of the data structures is not achievable except for a small set of kernel structures. B. Carrier and J. Grand in Kornblum (2007) discuss a strategy for robust address translation by incorporating invalid pages and paging file to improve the completeness of the analysis.

In this paper, instead of relying on the signatures of the structures, we try to determine some of the actions that a process has performed during its course of execution and by correlating these actions with each other and the process source code we will be able to extract a partial execution history of the process.

3. Approach

In this section, we lay out the principles underlying our approach to the forensic analysis of stack leftovers. What makes this approach possible is the way the stack operates in the course of program execution. The stack mechanism is used in most of the prevalent operating systems to make structured programming possible. For each function call made by a process, a stack frame is created and stored on the stack. The stack frame contains the parameters passed to the function, the return address, the previous value of the EBP register and the local variables of the function. These function call traces enclose the history of what a process has done during its course of execution.

After a function returns, the stack pointer is moved down to point to the previous stack. However, returned function stack frame still resides in the memory until another call is made by the process, and the stack grows up enough to overwrite the frame. The depth of the stack at each point of the execution depends on the number of nested function calls that are made by the process as well as the length of each stack frame. Due to the fact that the depth of the stack has arbitrary values during the execution, a large number of previously called function stack frames stay on top of the stack untouched or partially overwritten. Moreover, current software engineering best practices encourage the implementation of a service through long chain of function calls with each component serving some part of the requested service. This fact intuitively enforces our doctrine.

The correlation of the stack with the program source reveals the execution history of the program in terms of function call chains. We have developed modeling techniques, and the system that makes this approach possible. As shown in Fig. 1, the physical image acquired from the system under analysis is parsed to retrieve the process executable code

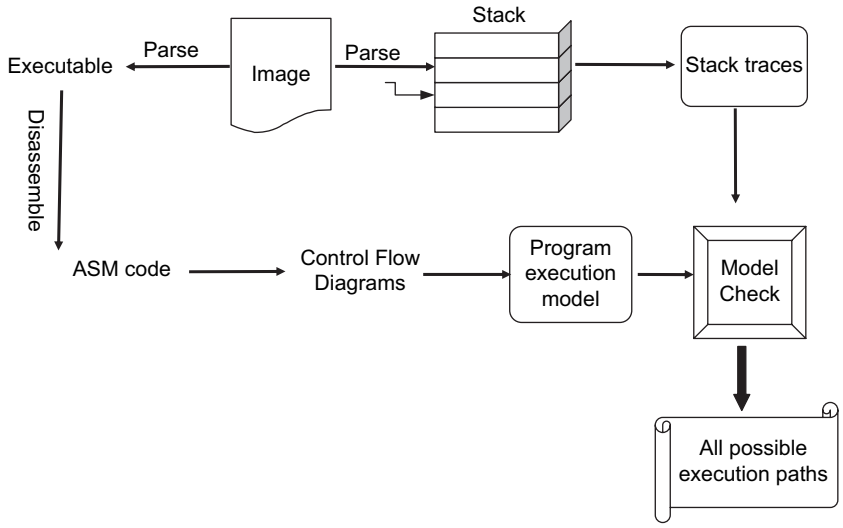


Fig. 1 – Our approach. The program model is verified against stack residues ADM model.

and thread stacks. The stack frames are extracted by analyzing the thread stacks. The extracted executable is analyzed to produce the control flow graph of each function and all the resulting CFGs are combined to form an abstract model of the program execution. The stack frames and their interrelations are modeled using the ADM process logic. The created model is verified against the generated program models using the tableau based proof system introduced in Debbabi et al. (2003). The next section discusses the details involved in this process.

Some of the advantages of this technique include:

- The analysis is performed on the extracted assembly code of the process from the memory and there is no need for the external provision of the source code or executable. This feature overcomes the anti-forensics that hide the executable in the filesystem by hooking operating system APIs.
- The technique integrates the formal analytical power of process logic and program models to retrieve the execution history of the process. This feature bestows the precision required in most jurisdiction systems from the digital investigation.
- As stated before, the result of the analysis could reveal important fact about what a process has done rather than what is currently existing in the memory. This is of paramount importance to forensics investigation since the final goal of the forensics science is to discover what has been done during the incident.
- The application of process logic instead of other modeling approaches such as state machines and PDAs to model the stack properties provides for higher performance in the analysis. The stack residues are naturally constraints on the execution history of the process. This is while, the state machine based modeling approaches are most effective when modeling the behavior of a whole system rather than the properties of the system. Using process logic favors the analysis by providing a means to model the constraints (in our case stack constraints) rather than developing a complete system.

4. Modeling the process and the stack traces

In this section, we detail our approach. The approach consists of five phases. First, we generate the CFG of all functions of the program. Second, the CFGs are transformed into finite state machines. Third, the finite state machines are combined to form a PDS system. The resulting PDS models the program execution in terms of function calls and returns made by the program. Fourth, we model the stack residues as a process logic property using ADM process logic (Debbabi et al., 2003). Fifth the resulting stack property is verified against the execution traces which can be generated by the PDS model of the program. The verification in this step is done based on the tableau based proof system introduced for ADM logic in Debbabi et al. (2003). In the following sections, we elaborate on each phase. As an example, consider the program shown in Fig. 2. We use this program to clarify each phase. For simplicity, we chose a program written in the C language. However, it is important to note that since our approach only deals with function calls and returns, exactly the same procedure is applicable to the assembly code.

4.1. *Control flow graph*

A control flow graph (CFG) is a structure that characterizes possible execution paths in a program. Vertices of the graph contain one or more instructions of the program that execute sequentially. Edges in the graph show how control flow transfers between blocks.

Let f be a function in a program P. The control flow graph for f is denoted by $G_f = \langle V_f, E_f \rangle$ where V_f is the set of vertices and $E_f \subseteq V_f \times V_f$ is the set of edges. A vertex in G_f is a basic block.

Each $v \in V_f$ contains a sequential list of instructions in f satisfying the following properties: there is no control flow transfer into the middle of a basic block nor a transfer out of the middle of a basic block. In defining basic blocks, notice that

```
1.  #include <iostream>
2.  void op(int i);
3.  void h(int i, int j) {
4.      return;
5.  }
6.  void g(int i, int j) {
7.      return;
8.  }
9.  void b(int i, int j, int k, int l) {
10.     return;
11.  }
12.  void e(int i, int j, int k, int l) {
13.     op(2);
14. }
15. void a(int i, int j, int k, int l) {
16.     if (i == 49) {
17.         g(i,j);
18.         e(i,j,k,l);
19.         return;
20.     }else{
21.         h(i,j);
22.         return;
23.        }
24. }
25. void c(int i, int j, int k, int l) {
26.     b(i,j,k,l);
27.     return;
28. }
29. void d(int i, int j, int k, int l) {
30.     h(i,j,k,l);
31.     return;
32. }
33. void op(int i) {
34.     char input;
35.     printf("Input a value
                between 1, 2:\n");
36.     fflush(stdin);
37.     scanf("%c", &input);
38.        a(i,0,0,0);
39.     switch (input) {
40.         case '1':
41.             d(0,0,0,0);
42.             break;
43.         case '2':
44.             c(0,0,0,0);
45.             break;
46.     }
47.     return;
48. }
49. void inc(int i) {
50.     if (i < 2) {
51.         inc(i+1);
52.     } else {
53.         op(1);
54.         return;
55.     }
56. }
57. void main() {
58.     inc(0);
59. }
```

Fig. 2 – Sample program to analyze.

the call to a function is considered as a transfer of control out of the basic block and therefore, each basic block at most has one function call instruction. An edge $\langle v_j, v_k \rangle \in E_f$ if there exists a possible control flow from v_j to v_k.

The first step of our approach is the generation of a control flow graph of each function called in the program. As an example, the control flow graph of function op of the sample program is shown in Fig. 3.

Having the CFG of a function, we generate the local automata model of the CFG as discussed in Giffin (2006). The local automata model of a CFG is a finite state machine whose states represent nodes of the CFG and its transitions are defined based on the control flows among different nodes of the CFG. Below is the formal definition of the local automata model.

Suppose that F is the set of functions in program P, C is the set of function call sites in P, and $\theta(c)$ denotes the target function of call site c. The local automata model of function f with control flow graph of $G_f = \langle V_f, E_f \rangle$ is defined as follows:

Let $a \lhd v$ indicate that vertex $v \in V_f$ contains call site a. The local model for f is $A_f = \langle Q_f, \Sigma_f, \delta_f, q_f, F_f \rangle$ where:

- $Q_f = V_f$.
- $\Sigma_f = C_f \cup \{\epsilon\}$.
- $F_f = C_f$ where $C_f \subseteq C$.
- $q_f \in V_f$ is the CFG entry state.
- $F_f = \{v \in V_f | v \text{ is a CFG exit state}\}$.
- Function call transition: $\delta_f(p, a) = q$ if $a \lhd p$, $a \in C_f$, and $\langle p, q \rangle \in E_f$.
- ϵ-transition: $\delta(p, \epsilon) = q$ if $\langle p, q \rangle \in E_f$ and $\forall a \in C_f . \neg(a \lhd p)$

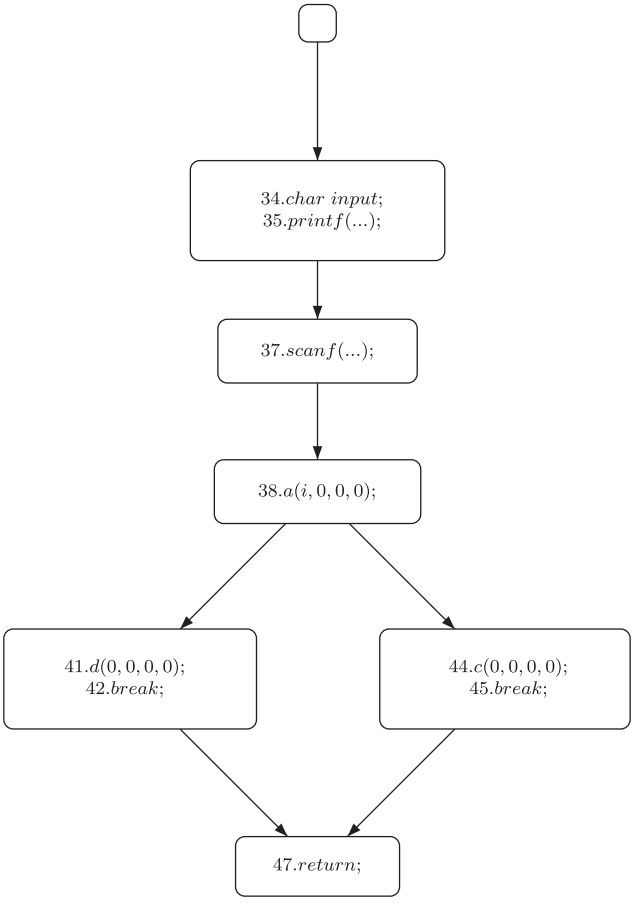


Fig. 3 – The control flow graph of op.

Please notice that we have changed the above model from the original version in Giffin (2006) by removing the system call transitions. This is due to the fact that, firstly since we are analyzing the kernel stack as well as the user land stack we do not need to restrict our analysis only to the user land system calls. Secondly, depending on the extent of the analysis, a stack trace analysis could expand to only the functions inside the program, the system calls, the library calls or even the low level kernel function calls. Therefore, we have introduced the concept of the end function calls which are a set of function names that are defined by the analyst to limit the depth of the analysis. The CFG of end function calls has only one state which is both an entry and an exit state. Intuitively, the local automata model of a CFG is a finite state machine whose states represent the nodes of the CFG and edges are either the name of a function called from the originating node, or ϵ.

As explained in Giffin (2006), the ϵ-reduction algorithm is performed on the local models to remove the ϵ transitions. This will increase the performance of the system since the ϵ edges are always traversed without consuming any symbol from the input. As an example, Fig. 4 depicts the local automata model of function op. Notice that in Fig. 4, we have included the line number in the transition names to differentiate among different calls to the same function. For the same reason, the definition of the local automata model of a CFG contains the concept of the function call site rather than the function name.

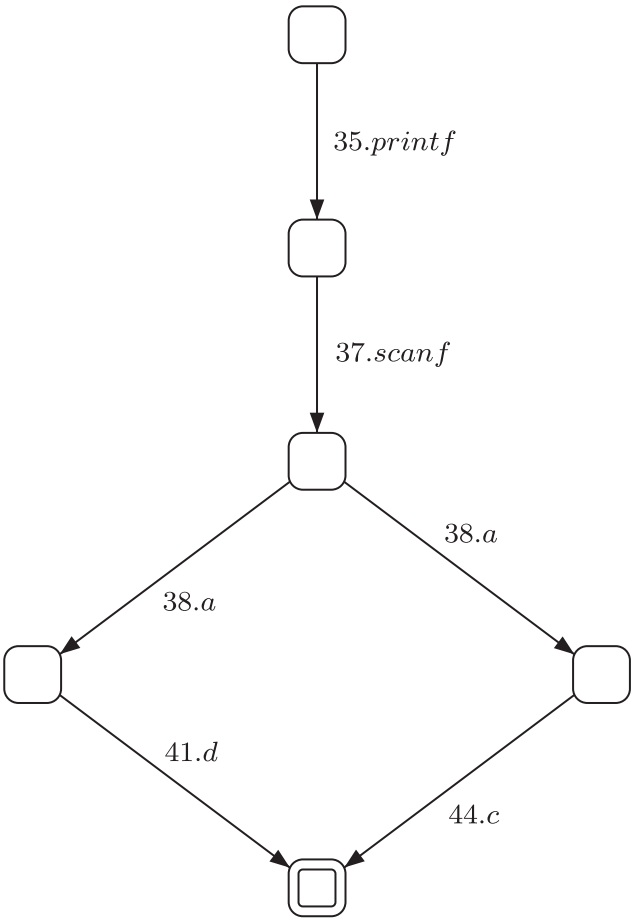


Fig. 4 – The local automata model of op.

Until now, we have modeled the execution of the program as a set of local state machines each representing the execution of a function in the program. However, in order to analyze the execution of a program as a whole, we have to combine the local state machine models into a global model. The resulting model should encompass all the possible control flows among the basic blocks of the program, while preserving the inter-procedural control flows. We have developed a modeling approach using Push Down System (PDS) that accurately models the execution of the program in terms of function calls and returns made by the program. The model maintains the inter-procedural execution flows.

A PDS is a triple $P = (Q, \Gamma, \sigma)$ where Q is the final set of control locations, Γ is the finite set of stack alphabets and $\sigma \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is a finite set of transition rules. The program execution in terms of the chain of function calls and returns made by the program is modeled using a PDS. We combine the local automata models of individual functions to form the PDS model of the whole program as follows:

Again suppose that F is the set of functions in program P, C is the set of function call sites in P, $\theta(c)$ denotes the target function of call site c. The combination of the local models of the functions of a program is defined as the PDS $P = (Q, \Gamma, \sigma)$ where:

- $Q = \cup Q_f$ for all $f \in F$.
- $\Gamma = C$ is the set of stack variables.
- Function call transition: $\sigma(p, \epsilon) = (q, r)$ if $\exists f \in F$, $c \in C$ such that $\delta_f(p, c) = r$, $q = q_{\theta(c)}$ where $q_{\theta(c)}$ is the entry state of the local automata model of $\theta(c)$.
- Function return transition: $\sigma(p, t) = (q, \epsilon)$ if $\exists f \in F$, $r \in Q$ such that $\delta_f(r, c) = q$, $p \in F_{\theta(c)}$ where $F_{\theta(c)}$ is the set of final states of the local automata model of function $\theta(c)$.

As an example, Fig. 5 shows the resulting PDS model of the program in Fig. 2. For clarity, in the diagram, the stack operations are represented as labels of edges. An edge labeled as a call site represents the push operation and an edge labeled as a bared call site represents the pop operation. Notice that in our analysis we have considered the *scanf* and *printf* functions as end functions. However, a more detailed analysis could involve modeling the function calls inside these functions.

4.2. *Modeling the stack*

The stack has a partial execution history of the program. However, based on the function call model of the program, the stack remnant could be interpreted in several different ways. A set of rules could be derived from the function call implementation using stacking mechanism as follows:

- If stack frame b is on top of stack frame c, then either c has called b or b has been called before c.
- The function call history should generate exactly the same stack trace and should not overwrite any of the currently existing stack frames.

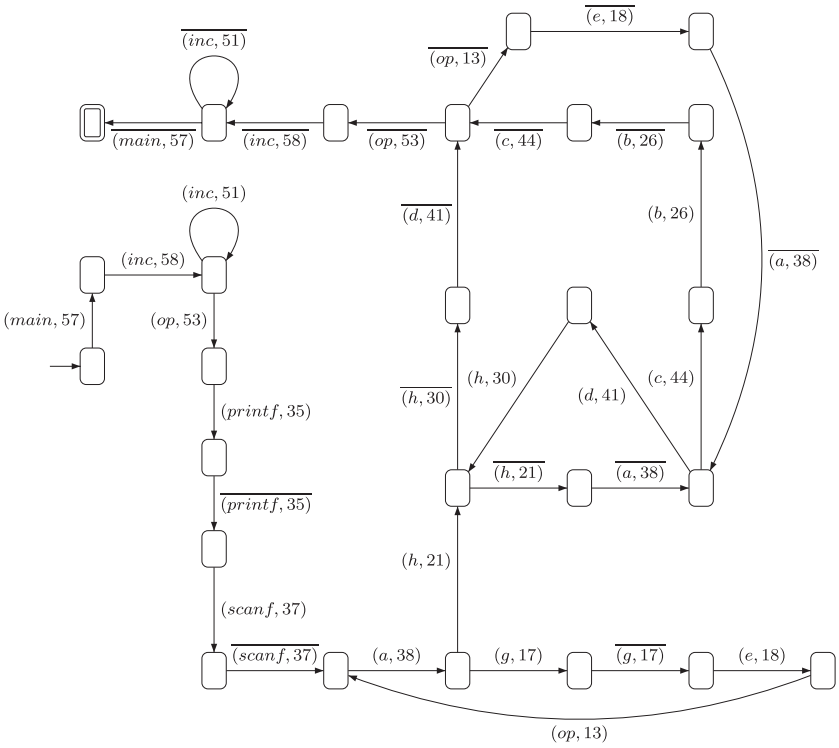


Fig. 5 – The PDS model of the program.

To model these properties for a stack trace, we use the ADM logic.

4.3. *ADM logic*

ADM (Debbabi et al., 2003) is a dynamic, linear, modal and trace based logic that has been developed for modeling and verification of security protocols. Through its compact and formal syntax and expressive semantic and due to it being a trace based logic rather than a state based process logic, one can specify a plethora of properties on the traces generated by the system. The syntax of the logic is based on patterns that are sequences of actions and pattern variables. A pattern is defined by the following grammar:

$$p ::= a \cdot p | x \cdot p | \epsilon$$

where ϵ stands for the empty pattern, a is an action and x is a pattern variable. Actions themselves may contain variables. In the sequel, the set of action variables is denoted by $\mathcal{V}_{\mathrm{a}}$, the set of pattern variables is defined by $\mathcal{V}_{\mathrm{p}}$, the set of message variables is represented by $\mathcal{V}_{\mathrm{m}}$, the set of session identifier variables is $\mathcal{V}_{\mathrm{ses}}$ and the set of step identifier variables is referred by $\mathcal{V}_{\mathrm{stp}}$. Here is an example of a pattern containing message variables, session identifier variables and pattern variables:

$$p = \mathrm{x_p} \cdot (\mathrm{x}_\alpha \cdot \mathrm{i}\ \mathtt{credit}(\mathrm{B}, \mathrm{M}, \mathrm{x_m})) \cdot \mathrm{y_p}$$

Intuitively, a pattern is an abstraction of a trace, where some actions or some attributes of an action are replaced by pattern and action variables, respectively. They are the basic elements used to specify formulae in the logic.

4.4. *Logic syntax*

Let X be a formula variable, then the set of logic formulae is obtained by the grammar given below:

$$\Phi ::= X | \neg\Phi | [p_1 \looparrowright p_2]\Phi | \Phi_1 \wedge \Phi_2 | \nu X \cdot \Phi$$

The symbols $\neg$ and $\wedge$ represent negation and conjunction, respectively, while $p_1 \looparrowright p_2$ is a modal operator indexed by the two patterns p_1 and p_2. The formula $\nu X \cdot \Phi$ is a recursive formula; the greatest fixed point operator ν binds all free occurrences of X in Φ. There is a syntactic restriction on the body of $\nu X \cdot \Phi$ stipulating that any occurrence of X in Φ must occur under the scope of an even number of negations. It is also assumed that the set of pattern variables in p_2 is included in the set of pattern variables in p_1 (no new variables appearing in p_2). For instance $[x \looparrowright x \cdot y]\nu X \cdot X$ is not a formula since {x, y} $\not\subseteq$ {x}.

From now on, $\mathcal{L}$ denotes the set of formulae of the logic and $\mathcal{V}$ is the set of formula variables (disjoint form $\mathcal{V}_{\mathrm{a}}$ and $\mathcal{V}_{\mathrm{p}}$). Furthermore, for convenience, we use the following standard abbreviations:

$$\begin{aligned}
&\mathrm{tt} \equiv \nu X \cdot X \\
&\mathit{ff} \equiv \mu X \cdot X \\
&\langle p_1 \looparrowright p_2 \rangle \Phi \equiv \neg [p_1 \looparrowright p_2] \neg \Phi \\
&\mu X \cdot \Phi \equiv \neg \nu X \cdot \neg \Phi [\neg X / X] \\
&\Phi_1 \vee \Phi_2 \equiv \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\
&\Phi_1 \rightarrow \Phi_2 \equiv \neg\Phi_1 \vee \Phi_2 \\
&\Phi_1 \leftrightarrow \Phi_2 \equiv \Phi_1 \rightarrow \Phi_2 \wedge \Phi_2 \rightarrow \Phi_1
\end{aligned}$$

where $\Phi[\Gamma/X]$ represents the simultaneous replacement of all free occurrences of X in Φ by Γ.

4.4.1. *Denotational semantics*

Suppose that Sub denotes the set of all possible substitutions σ such that:

$$\sigma \in [\mathcal{V}_p \to \mathcal{T}] \circ [\mathcal{V}_m \to \mathcal{M}] \circ [\mathcal{V}_a \to \mathcal{A}] \circ [\mathcal{V}_{ses} \to I_{ses}] \circ [\mathcal{V}_{stp} \to I_{stp}]$$

where $\mathcal{V}_p$ is the set of pattern variables, $\mathcal{V}_m$ is the set of message variables, $\mathcal{V}_a$ is the set of action variables, $\mathcal{V}_{ses}$ is the set of session identifier variables, $\mathcal{V}_{stp}$ is the set of step identifier variables, $\mathcal{M}$ is the set of messages, I_{ses} is the set of session identifiers, I_{stp} is the set of step identifiers and finally, $\mathcal{T}$ is the set of valid traces. The operation $\circ$ denotes function composition.

Env also denotes the set of all possible environments in $[\mathcal{V} \to 2^{\mathcal{T}}]$. Furthermore, we use $e[X \mapsto U]$ to denote the environment e' defined as follows:

$$e'(Y) = e(Y) \quad \text{if } Y \neq X$$
$$e'(X) = U$$

The semantics of formulae is given by the function:

$$[\![\_]\!]^{\_}_{\_}c : \mathcal{L} \times \mathcal{T} \times \text{Sub} \times \text{Env} \to 2^{\mathcal{T}}$$

defined inductively on the structures of formulae as shown in Table 1, where $t_\downarrow$ is the set of traces inductively defined as follows:

(i) $t \in t_\downarrow$
(ii) $t_1 \cdot a \cdot t_2 \in t_\downarrow \Rightarrow t_1 \cdot t_2 \in t_\downarrow$

Informally $t_\downarrow$ contains all subtraces that could be extracted from t by eliminating some actions from the beginning, from the middle and/or from the end of t. For instance, if $t = a \cdot b \cdot c$, then $t_\downarrow = \{\epsilon, a, b, c, a \cdot b, a \cdot c, b \cdot c, a \cdot b \cdot c\}$. The notation $t_\downarrow$ is introduced to simplify the presentation of the denotational semantics. Intuitively, given a trace t, the semantics of a formula will be all the traces in $t_\downarrow$ respecting the conditions specified by this formula.

Environments are used to give a semantics to the formula X and to deal with recursive formulae. Substitutions are internal parameters used to give a semantics to the formula $[p_1 \nrightarrow p_2]\Phi$. Given an environment e and a substitution σ, we say that a trace t satisfies Φ if:

$$t \in [\![\Phi]\!]^{t,\sigma}_e$$

Intuitively, the trace t satisfies the formula $[p_1 \nrightarrow p_2]\Phi$ if for all substitutions σ such that $p_1\sigma = t$, the new trace $p_2\sigma$ (the modified version of the trace t) satisfies the remaining part of the formula (Φ). In this respect, the notation $[p_1 \nrightarrow p_2]$ has principally two effects. First, the part p_1 allows us to verify if something has happened somewhere in the trace t. Second, the part p_2 allows us to modify the trace (delete some actions, substitute some actions by others, add some actions) in such a way the remainder of the formula (Φ) will be verified on the modified version of the trace described by p_2. Notice that the restriction on the used patterns ($\text{var}(p_2) \subseteq \text{var}(p_1)$) ensures that if $p_1\sigma = t$, then $p_2\sigma$ is a ground trace, that is, it does not contain any variables.

Table 1 – The denotational sementics of the logic

$$[\![X]\!]^{t,\sigma}_e = e(X)$$
$$[\![\neg\Phi]\!]^{t,\sigma}_e = t_\downarrow - [\![\Phi]\!]^{t,\sigma}_e$$
$$[\![\Phi_1 \wedge \Phi_2]\!]^{t,\sigma}_e = [\![\Phi_1]\!]^{t,\sigma}_e \cap [\![\Phi_2]\!]^{t,\sigma}_e$$
$$[\![[p_1 \nrightarrow p_2]\Phi]\!]^{t,\sigma}_e = \{u \in t_\downarrow \mid \forall \sigma' : p_1\sigma\sigma' = u \Rightarrow p_2\sigma\sigma' \in [\![\Phi]\!]^{p_2\sigma\sigma', \sigma' \circ \sigma}_e\}$$
$$[\![\nu X \cdot \Phi]\!]^{t,\sigma}_e = \nu f, \quad \text{where } \begin{cases} f : 2^{\mathcal{T}} \to 2^{\mathcal{T}} \\ U \mapsto [\![\Phi]\!]^{t,\sigma}_{e[X \mapsto U]} \end{cases}$$

4.4.2. *Logical modeling*

A stack frame contains the address from which the program execution should continue after the function is returned. Based on this address, except for dynamic function calls (using function pointers), both the callee and the caller and the exact address of the call site in the code are identifiable. Therefore, each stack frame in our trace represents a unique call site and since the PDS model also captures program flows based on the call site instead of the function name, each stack frame can be associated with a transition. The inclusion of the call site in modeling is due to the fact that a function could call another function in several different call sites. Accordingly, each stack frame in our trace is modeled as a triple (a, b, c) which represents function a being called by function b at call site c. Also the alphabets of the Dyke model are annotated by the call site as (a, c) showing the call to function a at call site c.

We define $a \to b$ to represent a calling b and $a < b$ with the meaning of a happened before b. To capture the stated properties, we need to be able to model the following logical statements:

- If the stack frame $C = (c, \text{call}_c)$ is before frame $B = (b, \text{call}_b)$ then the following logical statement is true:

  $$((C < B) \wedge (C \to B)) \vee (B < C) \wedge \neg(B \to C))$$

- Moreover, none of the stack frames currently existing on the stack should have been overwritten meaning that for each $c \in$ (*the set of stack frames*) the depth of the stack after calling the function representing c do not reach the depth of c.

To formally model the above statements, we need to formalize the statement of b happening before c and b calling c. These two properties could be modeled using ADM as follows:

$$b < c : \ < x_1 \cdot b \cdot x_2 \cdot b' \cdot x_3 \cdot c \cdot x_4 \nrightarrow \epsilon > tt$$

$$b \to c : \ \nu X < x_1 \cdot b \cdot c \cdot x_2 \nrightarrow \epsilon > tt \vee < x_3 \cdot b \cdot x_4 \cdot y_1 \cdot x_5 \cdot y_1' \cdot x_6 \cdot c \cdot x_7 \nrightarrow x_3 \cdot b \cdot x_4 \cdot x_5 \cdot x_6 \cdot c \cdot x_7 > X.$$

In order to model the third property, we define the combined execution trace:

Definition. If S is the stack trace and E represent an execution path that is accepted by the PDS model of the program, the *combined execution trace* is defined as follows: $\text{comb}(S, E) = S \cdot | \cdot E$

Essentially, the combined execution trace is the concatenation of both traces while separating the traces using the | symbol. The reason that we need to combine both traces is that with ADM logic we can only express properties on one single trace.

Using the combined execution trace of a system we can model the third property defined above concerning the persistence of stack frames as follows:

$$\nu X\langle z_1\cdot x_1\cdot|\cdot x_2\cdot z_1'\cdot x_3\looparrowright x_3\cdot \text{allow}\rangle(\nu Y\langle z_2\cdot x_4\cdot \text{allow}\looparrowright x_4\rangle Y \vee\langle z_3'\cdot x_5\looparrowright x_5\cdot\text{allow}\rangle Y\cdot)\wedge[z_4\cdot x_6\looparrowright x_6\cdot\text{allow}]X$$

Note that in the above formula, the trace variables starting with *x* are subtraces and the variables starting with *z* are single events. In analyzing each stack frame, the above formula adds the same number of allow constants at the end of the trace as the stack depth of the function frame. For each function call made after the return of the function representing the stack frame (z_1') the formula removes one *allow* from the end and for each function return it adds an allow at the end. This way, the formula does not allow the paths that overwrite the stack frame being analyzed.

In order to model the whole stack we add all the properties specified above in one formula by following the steps described below for every two consecutive stack frames starting from the bottom toward the top of the stack (stack limit):

1. Find the call and return states corresponding to the currently being analyzed frame:

$$\langle(z_1,-,c)\cdot-\cdot x_1\cdot|\cdot x_2\cdot(z_1,c)\cdot x_3\cdot(z_1',c)\cdot x_4\looparrowright x_3\rangle \times(\nu Z\langle(z_3,l)\cdot x_5\cdot(z_3',l)\cdot x_6\looparrowright x_5\cdot x_6\rangle Z\vee\text{empty}\cdot\text{tt})$$

   In the above formula, z_1 is the function representing the analyzed frame. The formula first chooses a call and return statements which appear after z_1' and both correspond to the same function call site. It proceeds by verifying if the right call and return are chosen by trying to match each call with its corresponding return statement and removing them from the trace.
2. Make sure that the stack frame is not overwritten by the following function calls:

$$\langle(z_1,-,c)\cdot-\cdot x_1\cdot|\cdot x_2\cdot(z_1,c)\cdot x_3\cdot(z_1',c)\cdot x_4\looparrowright x_4\rangle \times(\nu Y\langle z_4'\cdot x_7\looparrowright x_7\cdot\text{allow}\rangle Y\cdot\vee\langle z_4\cdot x_8\cdot\text{allow}\looparrowright x_8\rangle Y\cdot)$$

3. Model the properties. The properties consist of $z_1\rightarrow z_2$ and $(z_2\langle z_1\wedge\neg z_2\rightarrow z_1)$. Using the previous formulae, we have:

$$z_1\rightarrow z_2\equiv\nu U\langle(z_1,z_2,c)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_2\cdot(z_1,c)\cdot(z_2,-)\cdot x_9\looparrowright\epsilon\rangle \times\text{tt}\vee\langle(z_1,z_2,c)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_2\cdot(z_1,c)\cdot x_{10}\cdot(z_3,d)\cdot x_{11} \times\cdot(z_3',d)\cdot x_{12}\cdot(z_2,-,-)\cdot x_9\looparrowright(z_1,z_2,c)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_2 \times\cdot(z_1,c)\cdot x_{10}\cdot x_{11}\cdot x_{12}\cdot(z_2,-)\cdot x_9\rangle U$$

$$z_2\langle z_1\equiv\langle z_1\cdot z_2\cdot x_1\cdot|\cdot x_{12}\cdot z_2\cdot x_{13}\cdot z_2'\cdot x_{14}\cdot z_1\cdot x_{15}\looparrowright\epsilon\rangle\text{tt}$$

4. In order to be able to specify interrelationships among two consecutive stack frames, we have to locate each function call representing the stack frame in the trace. The first function call is located in step one. Therefore it remains for the second stack frame (b). The call site location for b should be preserved for the specification of properties among the next two consecutive frames (b, c). This is while the variable substitutions do not propagate through the iterations in the fix point formulae. Hence we will put some markers in the trace to mark the call site that we have found for b and in the next iteration, we remove them. Since the context specifying the location of b depends on the property we specify, depending on which property we want to describe in the previous step, we put the markers separately. In the following formulae ↑ and ↓ represent the before and after markers, respectively.

$$\text{if } z_1\rightarrow z_2:\ \langle(z_1,z_2,-)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_{16}\cdot(z_2,-)\cdot x_9\looparrowright(z_2,-,-)\cdot x_1\cdot|\cdot x_{16}\cdot\uparrow\cdot(z_2,-)\cdot\downarrow\cdot x_9\cdot\text{allow}\rangle$$

$$\text{if } z_2\langle z_1:\ \langle(z_1,z_2,-)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_{12}\cdot(z_2,-)\cdot x_{17}\looparrowright(z_2,-,-)\cdot x_1\cdot|\cdot x_{12}\cdot\uparrow\cdot(z_2,-)\cdot\downarrow\cdot x_{17}\cdot\text{allow}\rangle$$

Note that the reason we add the allow at the end of the formula is that the depth of the next stack frame that will be analyzed in the next iteration is one more than the current stack frame and therefore the maximum length of the function call chain can increase by one. Moreover we have removed the first frame from the stack to continue the specification of the properties for the next two consecutive frames.

As stated before, the markers should be removed from the trace in the next iteration. Accordingly we will add the formula to remove the markers from the trace at the beginning of the iteration after locating the related call site.

$$\langle(z_1,-,-)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_2\cdot\downarrow\cdot(z_1,-)\cdot\uparrow\cdot x_{20}\looparrowright(z_1,-,-)\cdot(z_2,-,-)\cdot x_1\cdot|\cdot x_2\cdot(z_1,-)\cdot x_{20}\rangle$$

To model the whole stack, we combine all the formulae using proper operators ($\vee$, $\wedge$). However, in order to reduce the complexity of the formula we define several macros to represent each step in the algorithm. In defining macros, it is important to specify the trace and stack variables which are used across several formulae as arguments in the macro.

1. Find the call and return states corresponding to the currently being analyzed frame.

$$\text{FindCallCite}(a,d,e)\equiv\langle(a,-,y_1)\cdot x_1|\cdot d\cdot(a,y_1)\cdot e\cdot(a',y_1)\cdot x_2\looparrowright e\rangle \times(\nu Z\langle(z_1,y_2)\cdot x_3\cdot(z_1',y_2)\cdot x_4\looparrowright x_3\cdot x_4\rangle Z\vee\text{empty}\cdot\text{tt})$$

2. Make sure that the stack frame is not overwritten by latter function calls.

$$\text{NotOverWriten}(a,d,e)\equiv\langle(a,-,y_1)\cdot x_1\cdot|\cdot d\cdot(a,y_1)\cdot e\cdot(a',y_1)\cdot x_2\looparrowright x_2\rangle(\nu Y\langle(z_2',-)\cdot x_3\looparrowright x_3\cdot\text{allow}\rangle Y\cdot\vee\langle(z_2,-)\cdot x_4\cdot\text{allow}\looparrowright x_4\rangle Y\cdot)$$

3. Model the properties.

$$a\rightarrow b\equiv\text{Call}(a,b,d,e)\equiv\nu U\langle(a,b,y_1)\cdot(b,-,y_2)\cdot x_1\cdot|\cdot d\cdot(a,y_1)\cdot(b,y_2)\cdot e\looparrowright\epsilon\rangle\text{tt}\vee\langle(a,b,-)\cdot(b,-,y_2)\cdot x_1\cdot|\cdot d\cdot(a,y_1)\cdot x_2\cdot(z_1,y_3)\cdot x_3\cdot(z_1',y_3)\cdot x_4\cdot(b,y_2)\cdot e\looparrowright(a,-,y_1)\cdot(b,-,y_2)\cdot x_1\cdot|\cdot d\cdot(a,y_1)\cdot x_2\cdot x_3\cdot x_4\cdot(b,y_2)\cdot e\rangle U.$$

$$b<a\equiv\text{Before}(a,b,e,f)\equiv\langle(a,-,-)\cdot(b,-,y_2)\cdot x_1\cdot|\cdot f\cdot(b,y_2)\cdot x_2\cdot(a,-)\cdot e\cdot(a',-)\cdot x_3\looparrowright\epsilon\rangle\text{tt}.$$

4. Mark call site locations.

$$\text{if } a \to b : \text{MarkIfCall}(a,b,e) \equiv \langle (a,b,-)\cdot(b,z_1,y_1)\cdot x_1 \cdot | \cdot x_2 \cdot (b,-)\cdot e \nrightarrow (b,z_1,y_2)\cdot x_1\cdot|\cdot x_2\cdot\uparrow\cdot b\cdot\downarrow\cdot e\cdot \text{allow}\rangle$$

$$\text{if } b < a : \text{MarkIfBefore}(a,b,f) \equiv \langle (a,-,-)\cdot(b,z_1,y_1)\cdot x_1\cdot|\cdot f \cdot (b,y_1)\cdot x_2 \nrightarrow (b,z_1,y_1)\cdot x_1\cdot|\cdot f\cdot\uparrow\cdot(b,y_1)\cdot\downarrow\cdot x_2\cdot\text{allow}\rangle$$

5. Unmark call site locations.

$$\text{Unmark}(a,g) \equiv \langle (a,z_1,y_1)\cdot x_1\cdot|\cdot g\cdot\downarrow\cdot(a,y_1)\cdot\uparrow\cdot x_2 \nrightarrow (a,z_1,y_1) \cdot x_1\cdot|\cdot g\cdot(a,y_1)\cdot x_2\rangle$$

Using the defined formulae, the whole stack specification could be modeled as follows:

$$\nu X\cdot\text{Unmark}(a,d)(\text{FindCallCite}(a,d,e)\wedge\text{NotOverWrite}(a,d,e) \wedge((\text{Call}(a,b,d,f)\wedge\text{MarkIfCall}(a,b,f)X\cdot)\vee(\text{Before}(a,b,g) \wedge\text{MarkIfBefore}(a,b,g)X\cdot)))$$

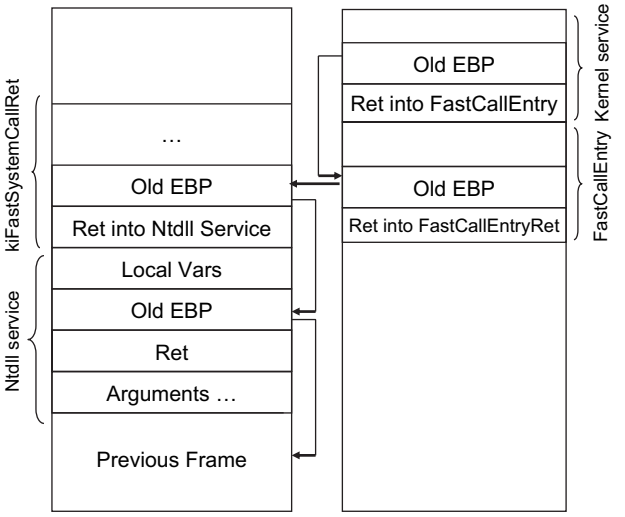


Fig. 6 – Windows thread stacks during the execution of a kernel service.

5. Windows architecture and memory layout

In this section, we summarize the stacking mechanism used in Windows. Knowing the functionality of Windows function call mechanism, the reader will gain a better understanding of the importance of the stack analysis in a digital forensics stack investigation. Windows operating system executes in two modes: user mode and kernel mode. Components that execute in the user mode are system support processes, service processes, user applications and environment subsystem server processes.

Under Windows, user applications do not call the native Windows operating system services directly; all the calls from user applications are redirected through a chain of subsystem Dynamic-Link Library (DLL) calls to the appropriate internal Windows system service calls. The kernel mode components of Windows include Windows executive, which provides for the primary operating system services, Windows kernel, device drivers, Hardware Abstraction Layer (HAL), and the windowing and graphics system.

As stated before, user mode processes cannot directly call services in kernel. The DLL *ntdll.dll* exports two sets of functions that are mostly wrappers for services inside the kernel and start with NT or ZW. Except for the functions that are handled inside *ntdll.dll* such as *NtCurrentTeb(...)*, which performs a purely user land operation, *ntdll.dll* exported functions are routed to a function with the same name in the *ntoskrnl.exe* (Schreiber, 2001). The routing mechanism that is performed by the system consists of switching the CPU from user mode to kernel mode, locating the service inside the kernel, copying the function parameters from user land stack to kernel land and executing the related service inside the kernel.

In Windows, each user application thread has at least two stacks, one in user land and one in kernel land. Accordingly, all functions that are called during the execution of the thread have a stack frame either in kernel land stack or the thread's user land stack depending on the mode they are executed under. In order to locate a service inside the kernel, Windows uses the service descriptor table that is located at the address of *KeServiceDescriptorTable* symbol. This table contains the address of each service and the number of arguments that should be passed to the function inside two arrays of the same length. A call to *ntdll.dll* functions loads a service number, which is an index into the arrays kept by the service descriptor table, and executes the *sysenter* command (Butler and Hoglund, 2005). This command changes the execution mode from user land to the kernel land and calls *kiFastSystemCallEntry*. This function locates the service address and jumps to the beginning of it. Fig. 6 shows the threads stack during the execution of a kernel service.

On returning from a kernel service, *sysexit* command is called. This command switches the execution mode from kernel mode to user mode and jumps to the *kiFastSystemCallRet* that simply returns from the user land *ntdll.dll* function call to the caller.

The provided kernel services can also be called by the drivers and other kernel modules. In this case, the calling is performed either in the context of a user application thread that has requested a service from the driver or in the context of system thread if the driver creates its own execution thread by calling *PsCreateSystemThread*.

The addresses of the kernel and user stacks are stored in the kernel structures. For each running process, there exists an *_eprocess* block in the kernel that contains the head of a link list of *_kthread* structures. Each *_kthread* structure represents one executing thread of the process. Fig. 7 depicts this structure. The structure fields *InitialStack*, *StackLimit* and *KernelStack* contain the stack base, the largest address that the stack can extend to, and the current position of the stack pointer, respectively. The *teb* field in the *_kthread* structure points to a structure of type *_TEB* that contains the user land information about a thread including the user stack. The first field in this structure is of type *_nt_tib*. The fields *StackBase* and *StackLimit* in this structure contain the stack base and the largest address the stack can extend to, respectively.

```
0:001> dt _kthread ntdll!_KTHREAD
+0x000 Header           : _DISPATCHER_HEADER
+0x010 MutantListHead   : _LIST_ENTRY
+0x018 InitialStack     : Ptr32 Void
+0x01c StackLimit       : Ptr32 Void
+0x020 Teb              : Ptr32 Void
+0x024 TlsArray         : Ptr32 Void
+0x028 KernelStack      : Ptr32 Void
+0x02c DebugActive      : UChar
            .....
+0x167 AutoAlignment    : UChar
+0x168 StackBase        : Ptr32 Void
+0x16c SuspendApc       : _KAPC
+0x19c SuspendSemaphore : _KSEMAPHORE
+0x1b0 ThreadListEntry  : _LIST_ENTRY
+0x1b8 FreezeCount      : Char
+0x1b9 SuspendCount     : Char
+0x1ba IdealProcessor   : UChar
+0x1bb DisableBoost     : UChar
```

Fig. 7 – The *_kthread* structure contains the address of the stack and the address of the top of the stack.

As stated before, our approach can work on assembly code. Moreover, the system we have developed is capable of extracting executable code from the memory eliminating the need for supplying the program executable based on the following technique. Files in Windows are mapped to the memory in two different ways: cached files and executables. The cached files are managed by Windows cache manager using structures of type *_VACB*. The executable files in the memory are mapped using the prototype PTE (Page Table Entry). The *SectionObject* field in *_eprocess* structure points to an object of type *section*. This object is represented by *_section_object* structure. For process executables, the *segment* field in this structure points to a structure of type *_segment* (although windbg describe it as *_segment_object*). The *_segment* structure and the process for extracting the first page of an executable are shown in Fig. 8. The *PrototypePte* field in this structure points to the beginning of the prototype page table for the process. This table is essentially a page table. Each entry contains the page frame number (physical page) of the physical memory that the corresponding page (virtual page) is mapped to. The executable code can be extracted by following each of these entries and copying the content of the pointed page frame.

6. Implementation and testing

The stack analyzer has been developed in a previously implemented Windows memory forensics environment. The environment enables the investigator to add analysis plugins, introduce new data structures and view the result of parsing a chunk of memory based on the introduced structures. The stack analyzer parses the stack based on two stack tracing techniques:

```
kd> dt _segment 0xe44db000
nt!_SEGMENT
   +0x000 ControlArea: 0x86c7fc10 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : 0x6f5
   +0x008 NonExtendedPtes  : 0x6f5
   +0x00c WritableUserReferences : 0
   +0x010 SizeOfSegment    : 0x6f5000
   +0x018 SegmentPteTemplate : _MMPTE
   +0x020 NumberOfCommittedPages : 0
   +0x024 ExtendInfo       : (null)
   +0x028 SystemImageBase  : (null)
   +0x02c BasedAddress     : 0x00400000
   +0x030 u1               : __unnamed
   +0x034 u2               : __unnamed
   +0x038 PrototypePte     : 0xe44db040 _MMPTE
   +0x040 ThePtes          : [1] _MMPTE

kd> dd 0xe44db040
e44db040  2a73d121 80000000 25a40121 00000000
e44db050  25a01121 00000000 25e42121 00000000
e44db060  3a24e121 00000000 1e1ec121 00000000
e44db070  26245121 00000000 25e06121 00000000
e44db080  00000460 86c7fc60 00000460 86c7fc60
e44db090  25dc8121 00000000 26209121 00000000
e44db0a0  25e4a121 00000000 2610b121 00000000
e44db0b0  2630c121 00000000 262cd121 00000000

kd> !dc 2a73d000
#2a73d000 00905a4d 00000003 00000004 0000ffff MZ..............
#2a73d010 000000b8 00000000 00000040 00000000 ........@.......
#2a73d020 00000000 00000000 00000000 00000000 ................
#2a73d030 00000000 00000000 00000000 00000120 ............ ...
#2a73d040 0eba1f0e cd09b400 4c01b821 685421cd ........!..L.!Th
#2a73d050 70207369 72676f72 63206d61 6f6e6e61 is program canno
#2a73d060 65622074 6e757220 206e6920 20534f44 t be run in DOS
#2a73d070 65646f6d 0a0d0d2e 00000024 00000000 mode....$.......
```

Fig. 8 – *_segment* Structure and extraction of executable content.

- As it is shown in Fig. 6, the OLD_EBP field on the stack holds the address of the previous frame OLD_EBP. In this way, stack frames are chained together and the stack parser follows this chain to correctly identify each stack frame. However, some compilers tend to use the EBP pointer within the function as a general purpose register. While this can optimize register utilization, it makes it impossible to trace back the stack by following the EBP chain.
- A stack frame can be identified by looking for return addresses that point to right after a call instruction. In this technique the stack will be traversed word by word testing which address is pointing to an instruction after a call instruction.

Using these two techniques, we are able to retrieve the addresses of call instructions as well as the arguments passed to the function. On the other hand, based on the technique discussed in the previous section, the parser extracts the executables from the memory. In order to disassemble the executable and generate the PDS model, we use IDAPro (The IDA pro, 2007). We have developed a plugin in IDAPro that generates the CFGs, local automata and PDS model of the program using the SDK provided by IDAPro.

Having the addresses (call sites) on the stack and the PDS model, the analyzer proceeds to verify the query specified

```
stack_trace!main+0xb [C:\stack trace\stack_trace.cpp @ 58]: 004012db
e83efdffff       call    stack_trace!ILT+25(?incYAXHZ) (0040101e)

stack_trace!inc+0x16 [C:\stack trace\stack_trace.cpp @ 51]: 004012a6
e873fdffff       call    stack_trace!ILT+25(?incYAXHZ) (0040101e)

stack_trace!inc+0x16 [C:\stack trace\stack_trace.cpp @ 51]: 004012a6
e873fdffff       call    stack_trace!ILT+25(?incYAXHZ) (0040101e)

stack_trace!inc+0x22 [C:\stack trace\stack_trace.cpp @ 53]: 004012b2
e853fdffff       call    stack_trace!ILT+5(?opYAXHZ) (0040100a)

stack_trace!op+0x63 [C:\stack trace\stack_trace.cpp @ 41]: 00401243
e8dbfdffff       call    stack_trace!ILT+30(?dYAXHHHHZ) (00401023)

stack_trace!d+0x11 [C:\stack trace\stack_trace.cpp @ 30]: 004011c1
e83ffeffff       call    stack_trace!ILT+0(?hYAXHHZ) (00401005)

stack_trace!a+0x2f [C:\stack trace\stack_trace.cpp @ 18]: 0040112f
e8fefeffff       call    stack_trace!ILT+45(?eYAXHHHHZ) (00401032)

stack_trace!e+0xb [C:\stack trace\stack_trace.cpp @ 13]: 004010eb
e81afffffff      call    stack_trace!ILT+5(?opYAXHZ) (0040100a)

stack_trace!op+0x75 [C:\stack trace\stack_trace.cpp @ 44]: 00401255
e8ddfdffff       call    stack_trace!ILT+50(?cYAXHHHHZ) (00401037)

stack_trace!c+0x19 [C:\stack trace\stack_trace.cpp @ 26]: 00401189
e8aefeffff       call    stack_trace!ILT+55(?bYAXHHHHZ) (0040103c)
```

Fig. 9 – Extracted stack traces.

before to find the possible execution path. Please note that the verification proceeds by first generating a possible execution path using the PDS model and then verifying it against the properties specified by the query. Which ever path that satisfy the query, could have been taken by the process at the time of the incident.

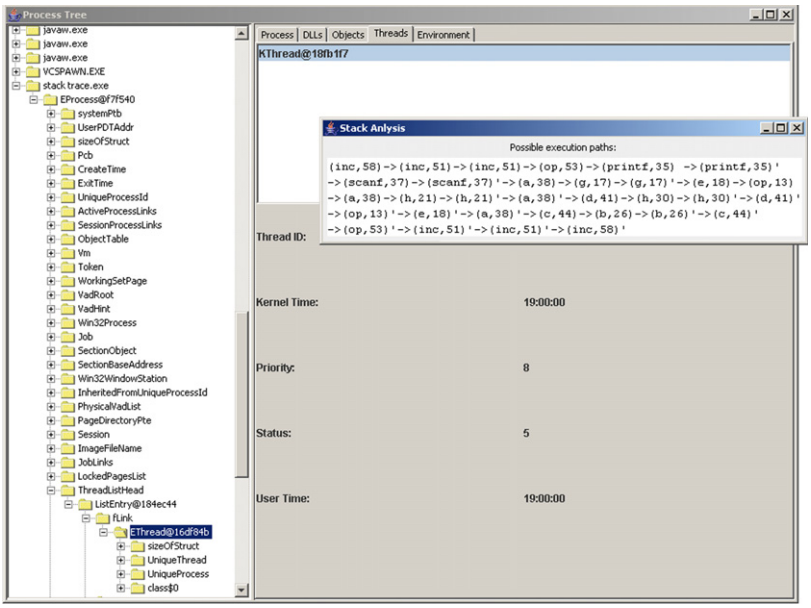
As a sample scenario, we have analyzed the program in Fig. 2. The program is executed with inputs of 1 and 2, in order. For simplicity, we limited our analysis to the functions called directly by the program. In this case, since the program has only two noncritical calls to the functions that are defined outside the program, we would not benefit from analyzing the kernel stack or other DLLs function calls. This is while in real situations, a great deal of information can be extracted by going deep inside the kernel stack and correlate the stack traces with program and operating system code. For demonstration purpose, the program also creates a windbg (Microsoft, 2007) script file that could be executed in the debugger to show the name of the function calls on the stack. The script file is essentially a list *ln* commands each having a function call address, that was found on the stack, as their argument. The result of executing this script in a debug session of windbg attached to our sample program is shown in Fig. 9.

Submitting the logical statement created in the previous section, the system is able to completely generate the executed function chain as below:

```
(inc,58)->(inc,51)->(inc,51)->(op,53)->(printf,35)
->(printf,35)'->(scanf,37)->(scanf,37)'->(a,38)->
(g,17)->(g,17)'->(e,18)->(op,13)->(a,38)->(h,21)
->(h,21)'->(a,38)'->(d,41)->(h,30)->(h,30)'->
(d,41)'->(op,13)'->(e,18)'->(a,38)'->(c,44)->
(b,26)->(b,26)'->(c,44)'->(op,53)'->(inc,51)'->
(inc,51)'->(inc,58)'
```

7. Conclusion

In this paper, we presented a new forensic analysis technique by analyzing the thread stacks. The information that is retrieved using this technique can reveal what has been done by the thread. A great deal of information can be acquired by correlating this information with the source code or the assembly code using different static code analysis approaches. Moreover, the stack trace and stack residue retrieved by our approach could be considered as a log from system activities and could be correlated with other sources such as network logs, operating system logs, etc. It is important to notice that due to the fact that each possible execution path is generated before it is verified, our approach is susceptible to infinite loops. However, the loop detection techniques could be applied to reduce the effect of infinite loops.



The stack analyzer.

REFERENCES

Burdach M. Digital forensics of the physical memory.

Betz C. MemParser, <http://www.dfrws.org/2005/challenge/memparser.html>; 2005 [Visited on May 22, 2007].

Butler J, Hoglund G. Rootkits: subverting the Windows kernel. Addison-Wesley Professional; July 2005.

Debugging tools for Windows, <http://www.microsoft.com/whdc/devtools/debugging/default.mspx> [Visited on May 22, 2007].

Digital Forensic Research Workshop (DFRWS). Memory analysis challenge; 2005.

Debbabi M, Adi K, Mejri M. A new logic for electronic commerce protocols. Int J Theor Comput Sci (TCS) 2003;291(3):223–83.

Giffin JT. Model based intrusion detection system design and evaluation. PhD thesis; 2006.

Gladyshev P, Patel A. Finite state machine approach to digital event reconstruction. Digit Investig J 2004;1(2).

Gladyshev P, Patel A. Formalising event time bounding in digital investigations. Digit Investig J 2005;4(2).

Garner Jr GM. Kntlist, <http://www.dfrws.org/2005/challenge/kntlist.html>.

Hosmer C. Time lining computer evidence, <http://www.wetstonetech.com/f/timelining.pdf>; 1998 [Visited on May 22, 2007].

Kruse II WG, Heiser JG. Computer forensics: incident response essentials. Boston, MA: Addison-Wesley; 2002.

Kornblum J. Using every part of the buffalo in Windows memory analysis. Digit Investig J January 2007.

Leigland R, Krings AW. A formalization of digital forensics. Digit Investig J 2004;3(2).

Monroe K, Bailey D. System base-lining: a forensic perspective, <http://ftimes.sourceforge.net/Files/Papers/baselining.pdf>; 2003.

Peikari C, Chuvakin A. Security warrior. Sebastopol, CA: O'Reilly; 2004.

Schreiber S. Undocumented Windows 2000 secrets: a programmer's cookbook. Addison-Wesley Professional; May 2001.

Schuster A. Pool allocations as an information source in Windows memory forensics. In: International conference on IT-incident management & IT-forensics, 2006.

Stallard T, Levitt K. Automated analysis for digital forensic science: semantic integrity checking. In: 19th annual computer security applications conference, Las Vegas, NV, USA, December 2003.

Stephenson P. Modeling of post-incident root cause analysis. Int Digit Evid 2003;2(2).

The IDA pro disassembler and debugger, <http://www.datarescue.com/ida.htm> [Visited on May 22, 2007].

Tenable Network Security. [Visited on May 22, 2007].

Walters A, Petroni N. FATKit: a framework for the extraction and analysis of digital forensics data from volatile system memory.

Walters A, Petroni N. Volatools: integrating volatile memory forensics into the digital investigation process. Black Hat DC 2007; February 2007.

Ali Reza Arasteh is a M.Sc. student at the Department of Computer Science, Concordia University. He holds B.Sc. degree from Amirkabir University, Tehran, Iran. His main research interests are memory forensics analysis, log analysis and intrusion detection systems.

Mourad Debbabi is a Full Professor and the Associate Director of the Concordia Institute for Information Systems Engineering at Concordia University. He is also a Concordia Research Chair Tier I in Information Systems Security. He holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay University, France. He published several research papers in international journals and conferences on computer security, cyber forensics, formal semantics, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory and specification and verification of safety-critical systems. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Corporate Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France.