Building stack traces from memory dump of Windows x64

Yuto Otsuki*, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, Kazuhiko Ohkubo

NTT Secure Platform Laboratories, Japan

Keywords:
Memory forensics
Stack trace
Windows x64
Thread context

Abstract

Stack traces play an important role in memory forensics as well as program debugging. This is because stack traces provide a history of executed code in a malware-infected host and this history could become a clue for forensic analysts to uncover the cause of an incident, i.e., what malware have actually done on the host. Nevertheless, existing research and tools for building stack traces for memory forensics are not well designed for the x64 environments, even though they have already become the most popular environment. In this paper, we introduce the design and implementation of our method for building stack traces from a memory dump of the Windows x64 environment. To build a stack trace, we retrieve a user context of the target thread from a memory dump for determining the start point of a stack trace, and then emulate stack unwinding referencing the metadata for exceptional handling for building the call stack of the thread. Even if the metadata are unavailable, which often occurs in a case of malicious software, we manage to produce the equivalent data by scanning the stack with a flow-based verification method. In this paper, we discuss the evaluation of our method through comparing the stack traces built with it with those built with WinDbg to show the accuracy of our method. We also explain some case studies using real malware to show the practicability of our method.

© 2018 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Introduction

Malware is widely used for various cyber attacks. To fight against such attacks, forensic analysis is a conventional approach. There are two forensic approaches for infected computers: disk and memory. Disk forensics is mainly used for investigating persistent data left by the incident, while memory forensics is mainly focused on acquiring runtime information when the incident occurs.

Stack traces play an important role in memory forensics as well as program debugging. This is because stack traces provide a history of executed code in a malware-infected host, and this history could become a clue for forensic analysts to uncover the cause of an incident, i.e., what malware has actually done on the host. Several methods related to stack traces for memory forensics have been proposed (Hejazi et al., 2009; Arasteh and Debbabi, 2007; Pulley, 2013; Smulders, 2013; Pshoul, 2017). They mainly use the traditional technique of walking a chain of frame pointers constructed on the stack. Most of them also include another solution of scanning the stack for return addresses, as measures against frame-pointer omission. In addition, WinDbg (Microsoft, 2017c), Microsoft’s official debugger, can obtain stack traces from a crash dump-type memory dump.

However, these methods may obtain inaccurate stack traces from programs without both frame-pointer chains and symbols. That inaccuracy of these methods is particularly problematic to apply them to applications running on the x64 versions of Windows. Windows x64 adopts unique software conventions, called “x64 Software Conventions” (Microsoft, 2017d; Murakami, 2010; CodeMachine Inc, 2011; Momot, 2013). Under these conventions, no frame-pointer chains are constructed in a stack. Therefore, the traditional technique is ineffective against 64-bit applications. In addition, Windows x64 has an emulation layer called WOW64 to execute 32-bit applications. To build stack traces from such applications, we should recognize their actual execution contexts on the layer. Scan-based techniques and WinDbg can find return addresses regardless of the environment (x86 or x64); however, they also have problems. Scan-based techniques may misdetect ordinary function pointers in a stack area as return addresses. WinDbg strongly depends on debugging symbols. This is a fatal defect for use in malware analysis because most of its code

* Corresponding author.
E-mail address: otsuki.yuto@lab.ntt.co.jp (Y. Otsuki).

https://doi.org/10.1016/j.diin.2018.01.013
1 x64 is also known as AMD64, IA-32e, x86_x64 and so on. In this paper, we denote the 64-bit version of the x86 architecture as x64 in accordance with Microsoft documents.
regions have neither symbols nor any other metadata. Since x64 environments have already become the most popular environment, developing new methods for Windows x64 is absolutely necessary for incident response.

In this paper, we introduce the design and implementation of our method for building stack traces from a memory dump of the Windows x64 environment. To solve the above-mentioned problem, our method includes two major approaches: emulating stack unwinding and verifying actual return addresses found by scanning. Our method first obtains the last execution state of user context of each thread saved in memory by Windows. Since the context contains the last stack pointer, we can recognize the actual top of the stack. To obtain return addresses in the stack, our method interprets the metadata for exception handling on the memory dump. If the metadata are unavailable, which often occurs in a case of malicious software, we manage to produce the equivalent data by scanning the stack with a flow-based verification method. Our method extracts all candidates for a return address by using a conventional scan-based technique, and finds the correct return address from them based on control-flow analysis. Our method also includes a custom-tailed function based on walking frame-pointer chaining for a 32-bit application on the WOW64 layer.

We evaluated our method’s accuracy and practicability through experiments. We confirmed that our method could build stack traces of the x64 process with or without metadata and WOW64 process. We also confirmed that it could build stack traces more accurately than using just a conventional scan-based technique. Through experiments using real malware samples, we confirmed our method is effective against malware holding threads to wait for events, such as bot-type malware.

The contributions of this paper are as follows.

- We introduce a method of emulating stack unwinding to build a stack trace of each thread from only memory dump of Windows x64.
- We also introduce a flow-based verification method, which more precisely identifies return addresses than using only conventional scan-based techniques.
- We evaluated our method by comparing the stack traces built with it with those built with WinDbg to show the accuracy of our method. We also explain case studies using real malware to show the practicability of our method.
- We discuss our method’s limitations, discuss methods of disrupting our method, and present countermeasures against them.

**Background and problem**

In the Windows x86 environment, there are two types of functions, i.e., with or without using a frame pointer. In this section, we explain conventional stack-trace techniques for both types and a problem of these techniques.

**Stack trace for functions with frame pointer**

A function with a frame pointer generally pushes the current value of the frame-pointer register to the stack, e.g., `push ebp`. Then, the function updates it with the current value of the stack pointer, e.g., `mov ebp, esp`. Since the frame-pointer register always keeps pointing to the boundary between the local buffer and previously stored frame pointer, the code in the function is able to access its local variables or its function arguments stored in the stack via the frame-pointer register, e.g., `mov eax, [ebp+0x1c]`.

The stack-trace technique for this type of function obtains a call-stack based on the frame pointers stored in the stack. Fig. 1 illustrates a traditional stack-trace technique for the Windows x86 environment. As we mentioned above, the frame-pointer EBP points to the previous one. In addition, the return address to a function is stored right before the frame pointer in the stack. These facts allow us to make a chain of caller and callee functions through the stored frame pointers. By connecting the chains, we can construct a call-stack at the current execution state, i.e., the list of caller and callee functions in the correct order.

**Stack trace for functions without frame pointer**

A function without using a frame pointer is generated as a result of compiler optimization configuration. For example, in the case of Visual Studio, the compiler option for this is frame-pointer omission/`/Oy` (FPO) option. Since a compiler can precisely acquire the stack size when it generates code, it calculates the offset of each local variable from the top of the stack for instructions to access them. This mechanism allows us to access local variables without using the EBP as a frame pointer and use it as a general-purpose register.

Since no EBP chains exist in a call stack of FPO-applied functions, the technique we mentioned in Subsect. 2.1 does not work in this situation. While all Windows x86 platforms released after Windows XP Service Pack 2 disable FPO for all dynamic link libraries (DLLs) and executables (Microsoft, 2017a), malware or third-party binaries may use this optimization. In addition, there are functions that do not use a frame pointer for all Windows's official binaries, e.g., system call stubs and completely internal functions. The native 64-bit applications on all Windows x64 platforms generally conform to x64 Software Conventions (Microsoft, 2017d; Murakami, 2010; CodeMachine Inc, 2011; Momot, 2013). They do not construct any frame-pointer chaining in the stack. Debugging symbols are basically required to build stack traces of these applications.

To build stack traces for this type of functions even if symbols are unavailable, there are techniques (Hejazi et al., 2009; Arasteh and Debbabi, 2007; Pulley, 2013) for scanning the stack for return addresses and constructing a call-stack based on the found return addresses without depending on the frame pointers. These techniques collect StackBase and StackLimit from the thread environment block (TEB) as the highest and lowest addresses of the stack, respectively, and scans the memory area between them for addresses that satisfy the following conditions.

1. address pointing to executable memory area
2. address pointing to an address whose previous instruction is `call`

This subsection describes a problem of the existing methods for building stack traces. It also covers some implementation issues for Windows x64.
دریافت فوری متن کامل مقاله

امکان دانلود نسخه تمام متن مقالات انگلیسی
امکان دانلود نسخه ترجمه شده مقالات
پذیرش سفارش ترجمه تخصصی
امکان جستجو در آرشیو جامعی از صدها موضوع و هزاران مقاله
امکان دانلود رایگان ۲ صفحه اول هر مقاله
امکان پرداخت اینترنتی با کلیه کارت های عضو شتاب
دانلود فوری مقاله پس از پرداخت آنلاین
پشتیبانی کامل خرید با بهره مندی از سیستم هوشمند رهگیری سفارشات