Digital Investigation 24 (2018) S101–S110

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2018 Europe — Proceedings of the Fifth Annual DFRWS Europe

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

A B S T R A C T

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.

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[1] 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).

[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.

https://doi.org/10.1016/j.diin.2018.01.013

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-tailored 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+0xC]`.

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.

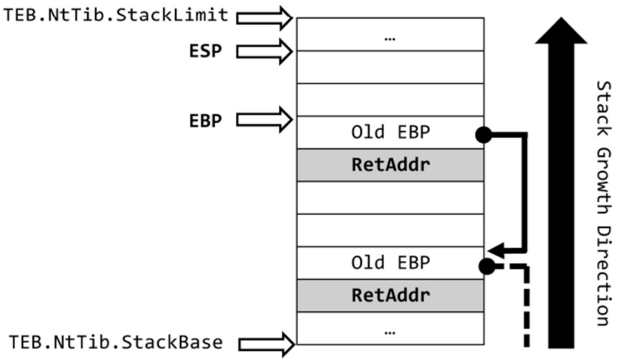

Fig. 1. Traditional stack-trace technique based on EBP chaining.

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`

Problem and issues

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

The main problem is, as far as our experimental results, the scan-based techniques included in the existing methods are potentially inaccurate. The Windows x64 environment has already become popular environment. In other words, functions without a frame pointer have already become popular. Since malware and released applications generally do not have symbols, we actually have no choice but to use scan-based techniques to build stack traces in the memory-forensic context. However, these techniques may misdetect ordinary function pointers and inactive return addresses in a stack area as active return addresses. In fact, we found many false positives in the result obtained by a scan-based technique in our experiment, described by Subsect. 5.5. Therefore, we have to establish a more accurate technique to build stack traces for functions without a frame pointer from just a memory dump.

There are also issues with implementing practical tools for Windows x64. Although many of the existing methods recognize the stack area through the TEB prepared by Windows, we can use an arbitrary memory area as a stack by letting the stack pointer point there. We have to know the area being used as the stack. We should also recognize execution contexts of 32-bit applications on the WOW64 layer. For these applications, we have to parse the emulation layer's context then build stack traces from the stack used by 32-bit applications.

Overview

This section gives a summary of our method to build stack traces from a memory dump of the Windows x64 environment.

Requirements

In this subsection, we explain requirements to implement building stack traces from a memory dump of the Windows x64 environment.

Implementing stack traces essentially requires (1) recognizing the actual range of the active stack used by the stack-owner thread, and (2) locating the position of each return address in the stack. A stack-trace technique retrieves return addresses from the stack to reconstruct the call-stack at the current execution state. As mentioned in Subsect. 2.2, the stack pointer used is needed to accurately obtain stack traces. In addition, a stack contains not only return addresses but also other values. We need to identify which stack elements are return addresses.

More concretely, to meet the requirements from a memory dump of the Windows x64 environment, implementing a stack trace requires conforming to the Windows x64-specific convention. Running processes on the Windows x64 environment are classified roughly into two types, as shown in Fig. 2.

(a) x64 process executing a 64-bit application on the native environment
(b) WOW64 process executing a 32-bit application on the WOW64 layer

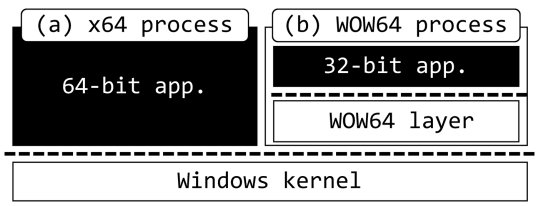

Fig. 2. Processes on Windows x64.

Since the context managements and stack usages are different for each process type, both implementations to meet requirements (1) and (2) are needed for the types (a) and (b), respectively.

Our approaches

This subsection covers our approaches to meet the requirements explained by the previous subsection.

We present two approaches for each process type. The approach for process (a) includes methods to implement requirements (1) and (2). These methods are denoted as (a-1) and (a-2), respectively. Similarly, the other approach for process (b) includes methods of implementing requirements (1) and (2). These methods are denoted as (b-1) and (b-2), respectively.

Approach for x64 process

We explain our approach for process (a). It obtains the last execution context from the kernel to meet requirement (1) and basically emulates stack unwinding based on metadata for exception handling for requirement (2). In addition, it includes another solution for requirement (2) for the case in which metadata are unavailable.

Method (a-1) obtains the last context of the target thread from the thread object corresponding with the thread. The context is saved by Windows for resuming its execution. It contains the last value of the registers, such as the RSP register pointing to the top of the stack, and the RIP register pointing to the next instruction. In contrast to StackLimit and StackBase, the RSP register indicates the location of the actual stack top. Even if the RSP points outside between them, the actual range of the stack can be regarded as the continuous pages from the location pointed to by the RSP register. We give details on how to obtain the context in Subsect. 4.1.

Method (a-2) includes two sub-methods depending on whether metadata for exception handling are available. The x64 processes do not use a frame pointer because they conform to x64 Software Conventions. Functions under the conventions provide metadata for exception handling. Therefore, if the metadata are available, we can extract return addresses in the stack by emulating stack unwinding according to the metadata. When the metadata are not available, we first extract all candidates for a return address by using a conventional scan-based technique, and find the correct return address from them based on control-flow analysis. The sub-methods for these cases are denoted as (a-2-x) and (a-2-y), respectively. The details of these sub-methods are given in Subsect. 4.2 and Subsect. 4.3, respectively.

Approach for WOW64 process

We also present our method for building stack traces of a 32-bit application on WOW64 layer by meeting the essential requirements. A thread on process (b) has two execution contexts of each 32-bit application and WOW64 layer. There are also two stacks for a thread. Methods (b-1) and (b-2) should obtain the execution context of a 32-bit application because 32-bit malware runs as a 32-bit application, as shown in Fig. 2.

The WOW64 layer is the key for our method. It has the last execution context of each thread in a 32-bit application similar to a thread object holding a context of each thread in a 64-bit native application. A 32-bit application on the WOW64 layer uses its stacks under the traditional convention. Thus, method (b-2) can use the traditional EBP-chaining-based technique for the WOW64 process. Subsection 4.4 covers the WOW64 layer and methods (b-1) and (b-2).

Execution flow

This subsection explains the execution flow of a practical use of our methods described above. We implemented our methods as a plugin for the Rekall memory analysis framework (Google Inc., 2017). The flow of the plugin has two phases, preparation and main. The main phase also has two cases for each process ((a) and (b)).

Preparation phase

In this phase, our plugin prepares for our main methods. Since the main methods are designed to obtain a stack trace of a thread with a user context, the plugin first enumerates thread objects (ETHREAD structure) included by a memory dump. We can use existing implementations such as `threads` plugin and `thrdscan` for Rekall. Second, the plugin detects whether the process owning the target thread is (a) or (b) by checking the Wow64Process member of the EPROCESS structure representing the process object. If the member holds the NULL pointer, the target process is (a). If the member holds a valid pointer, the target is (b). Finally, the plugin moves into the main phase ((a) or (b)) according to the process type.

Main phase (a): x64 process

In this phase, our plugin obtains a stack trace of a thread of x64 process. Method (a-1) obtains the last execution context of the target thread to recognize the actual stack range. After obtaining the context, our plugin starts locating each return address in the stack pointed to by the obtained RSP register. Method (a-2) checks whether the metadata for the current RIP are available or not and activates sub-method (a-2-x) or (a-2-y). These sub-methods locate the previous return address in the stack and updates the RIP and RSP values based on the assumption that the RIP is returned to the caller from the callee. Our plugin repeats calling method (a-2) until RSP reaches the bottom of the stack or RIP points to outside the code regions.

Main phase (b): WOW64 process

In this phase, our plugin obtains a stack trace of a thread of the WOW64 process. It first activates method (b-1) to obtain the context of a thread in a 32-bit application. Second, method (b-2) is activated and checks the EIP pointing to a system call stub. Since those stubs are not ordinary functions, method (b-2) parses their special usages of the stack area to obtain a return address from the stub. Finally, it starts extracting return addresses by walking through the EBP chaining.

Implementation details

This section details the implementation of our method.

(a-1) Obtaining a context of a thread running on the Windows x64 environment.
(a-2-x) Emulating stack unwinding based on metadata for exception handling.
(a-2-y) Locating the previous return address based on flow-based verification.
(b-1) and (b-2) Building a stack trace of a thread of the WOW64 process.

Obtaining user contexts of x64 process

This subsection covers how to obtain a user context of each thread owned by the x64 process. Note that operating systems (OSes) generally store a thread context when an event occurs, i.e., context switch, system call invocation, and interrupts. The OS saves the context of the current thread into memory to guarantee it resuming while handling such events.

Therefore, our method can obtain user-mode contexts of threads before returning such an event from memory. Specifically, threads in a waiting state must hold their last execution contexts. If the running thread invoked a system call and the OS kernel was handling it when a memory dump was obtained, we can also obtain the context of the thread.

The ETHREAD object holds the saved execution context in the Windows architecture regardless of the environment (x86 or x64). The structure for saving a context is called KTRAP_FRAME. The existing method (Pulley, 2013; Pshoul, 2017) also uses this structure to obtain the last EBP value for the Windows x86 environment. Although Windows x64 does not generally save all registers into KTRAP_FRAME (OSR Open Systems Resources and Inc., 2009), all registers are saved when user-to-kernel mode transition occurs (CodeMachine Inc, 2011).

We introduce how to obtain a user context of a target thread from a memory dump of the Windows x64 environment. Fig. 3 shows the structure of the ETHREAD and KTRAP_FRAME. The ETHREAD object contains a KTHREAD object as the Tcb member. The KTHREAD object holds a pointer to the last KTRAP_FRAME structure for the thread as the TrapFrame member. The ETHREAD.Tcb.PreviousMode member indicates whether the TrapFrame member has a user context or a kernel one (Microsoft, 2017e). Thus, our method obtains a user context from ETHREAD.Tcb.TrapFrame.

Emulating stack unwinding

The traditional use of the RBP register is completely optional under the x64 Software Conventions. By default, a function uses offsets from the top of the stack pointed to by the RSP register to refer to its arguments and local variables in the stack. The RBP register is used as a normal non-volatile general-purpose register. The function does not push the RBP into the stack. In other words, the stack has no frame-pointer chainings.

Instead, the PE32 + file has metadata for stack unwinding. This executable image format is an extended version of the conventional PE32 format for Windows x64. The metadata are embedded in the image file to support structured exception handling (SEH) provided by Windows and C++ exception handling. They indicate stack usages for each function in the image. The exception-handling mechanism unwinds the stack pointer and restores the registers based on the metadata. Therefore, in memory forensics, a stack trace can be obtained based on these metadata loaded into memory.

There are three types of structures for stack unwinding under the x64 Software Conventions: RUNTIME_FUNCTION, UNWIND_INFO, and UNWIND_CODE. In this paper, these structures are collectively called unwind information. The x64 Software Conventions include the definition of these structures and their

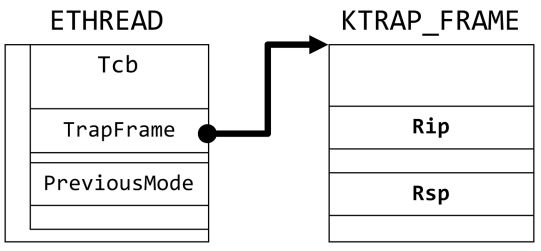

Fig. 3. TrapFrame held by ETHREAD object.

relationship. Fig. 4 illustrates the basic relationship pattern of these structures. Although we found some other patterns of unwind information including undocumented ones, the role of each structure is almost the same.

The RUNTIME_FUNCTION structure is prepared for each function in the image file. The .pdata section, called Exception Directory, includes an array of this structure. The structure holds three relative virtual addresses (RVAs). Two RVAs point to the beginning and end of the function and the third points to the UNWIND_INFO structure for the function.

The UNWIND_INFO structure represents information for unwinding. It contains an array of the UNWIND_CODE structure, and each UNWIND_CODE in the array indicates an operation in the function's prolog, which affects the RSP or other non-volatile registers. For example, the UWOP_PUSH_NONVOL code indicates that the prolog pushes a nonvolatile register. The UWOP_ALLOC_SMALL and UWOP_ALLOC_LARGE codes allocate space on the stack. If a function uses a frame pointer to dynamically allocate space on the stack under the x64 Software Conventions, UNWIND_CODEs for the function includes the UWOP_SET_FPREG code to specify a register used as a frame pointer.

Sub-method (a-2-x) interprets these data structures on a memory dump and emulates stack unwinding to obtain the previous return address. It also restores registers because they may be used as a frame pointer in accordance with UNWIND_CODEs. The procedure of sub-method (a-2-x) is as follows.

1. It obtains a base address of the region pointed to by the current RIP by referring to the process' tree of virtual address descriptors (VADs), which represents a memory map of the process' user space (Dolan-Gavitt, 2007).
2. It confirms that the current RIP points inside the PE32 + image by checking the signature of the Portable Executable (PE) header placed in the base address of the region.
3. It obtains the RVA of Exception Directory and calculates the directory's virtual address.
4. It obtains a RUNTIME_FUNCTION structure whose range contains the current RIP from Exception Directory.
5. It calculates a virtual address for the UNWIND_INFO pointed to by the UnwindData member of the obtained RUNTIME_FUNCTION.
6. It unwinds RSP and restores other registers based on each UNWIND_CODE included by the UNWIND_INFO.
7. If the unwind information chains other unwind information, this sub-method obtains the chained unwind information and repeats from step 4.2 until arriving at the last chained one.
8. It pops the previous return address from the stack after completing all unwind operations and sets the RIP up as the value.

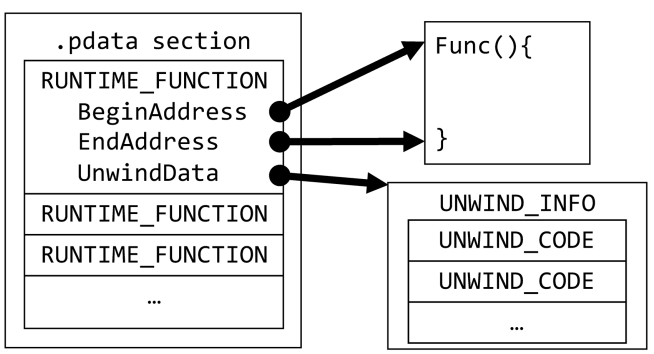

Fig. 4. Basic structure of unwind information.

Windows has another table called RtlpDynamicFunctionTable and several related APIs to support exception handling for dynamically generated code. Even if the PE signature is not found in step 2, unwind information may be available in the table. In this case, the following steps replace steps 3 and 4, respectively.

3'. This sub-method obtains a base address of ntdll.dll by referring to the VAD tree and calculates a virtual address for RtlpDynamicFunctionTable.
4'. It obtains a RUNTIME_FUNCTION structure from RtlpDynamicFunctionTable in the same manner as obtaining it from Exception Directory.

Flow-based verification for identifying return address

This section gives details of sub-method (a-2-y). The current RIP, which means the last obtained return address, often points to a code region without the PE header, i.e., dynamically generated code such as unpacked original code and injected code. Such a region has no metadata for exception handling. Sub-method (a-2-y) is activated if sub-method (a-2-x) fails to obtain metadata for exception handling for whatever reason. A typical example of this case is that the PE header is not detected at the top of the region and no metadata are found in RtlpDynamicFunctionTable. There are also exceptional cases in which metadata had not yet loaded on memory when the dump was obtained.

As stated above, the true caller definitely has one or more execution paths to reach the current RIP. Sub-method (a-2-y) first uses a conventional scan-based technique and marks the detected entries as candidates for the previous return address. It then finds the correct return address by verifying for each candidate based on the above-mentioned assumption.

We explain the procedure of this sub-method by Fig. 5 as an example. In this figure, there are two candidates, Pointer1 and Pointer2, inside the stack. The procedure is as follows.

(1) Sub-method (a-2-y) scans candidates from the current RSP, just like conventional techniques. In the case of Fig. 5, Pointer1 and Pointer2 are found.
(2) It obtains an instruction right before that pointed to by each candidate. It obviously must be a `call` instruction.
(3) It analyzes a control flow inside the function targeted by each obtained `call` instruction.
(4) It finds execution paths to reach the current RIP and adopts the candidate right after the `call` instruction targeting the function including the paths as the current one. In the case of Fig. 5, Pointer2 can be the correct return address.
(5) It sets the current RIP up as the adopted return address. It also sets RSP up as a virtual address pointing to the next entry of the return address, such as RSP′ in Fig. 5.

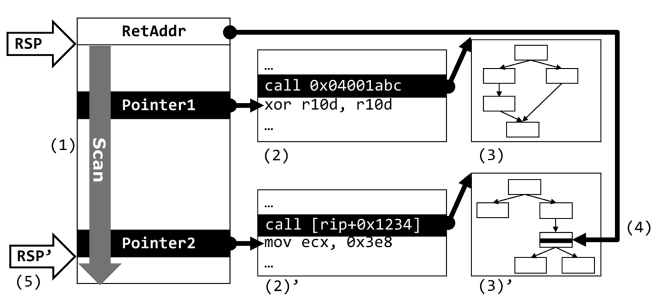

Fig. 5. Flow-based verification for identifying return address.

In step 3, the `call`'s target cannot be identified statically if the `call` is an indirect call via register, i.e., `call rbx`. Sub-method (a-2-y) cannot execute flow-based verification for such candidates. However, their candidates may reach the current RIP. Therefore, it adopts such candidates if no other candidates have reachable paths.

Stack trace of WOW64 process

In this subsection, we describe our stack-trace method for the WOW64 process and explain how to obtain user contexts of a 32-bit application on the WOW64 layer. We also describe obtaining return addresses from a 32-bit application's stack.

The WOW64 layer emulates the Windows x86 environment to execute a 32-bit application. The WOW64 process executes the 32-bit application in x86 mode. When a 32-bit application on the WOW64 layer invokes system call, the WOW64 layer switches the processor's mode to x64. The layer emulates the system call by invoking real system calls if needed. In this system-call emulation, the layer first saves the registers to memory as the execution context of a 32-bit application to restore them when the invoked system call is returned. In other words, the WOW64 layer's system-call handling is very similar to the OS's, as mentioned in Subsect. 4.1 Thus, we can address this first technical issue of obtaining a context by obtaining the register values from memory, just as with method (a-1).

Fig. 6 illustrates a path to reach the location of a user context held by the WOW64 layer from the ETHREAD object. A thread of the WOW64 process has two contexts of the WOW64 layer's and 32-bit application's. The 32-bit application's context is stored as a WOW64_CONTEXT structure in the layer's thread local storages (TLSs) (Microsoft, 2017b; Johnson, 2007). There are also two TEBs for each context. In this paper, we call the TEB for a 32-bit application TEB32 and that for a 64-bit native application TEB64. The WOW64 layer's TEB is TEB64, and it is pointed to by the Teb member of the ETHREAD object. TEB64 has a TlsSlots array for the TLS. As officially documented on the MSDN website (Microsoft, 2017b), the WOW64_CONTEXT structure is contained by an undocumented structure pointed to by slot 1 of the TLS. The WOW64_CONTEXT starts at offset 4 of that unknown structure. Sub-method (b-1) retrieves a user context of the 32-bit application by walking these structures.

A 32-bit application on the WOW64 layer uses its stacks in a conventional manner of Windows x86. Thus, an EBP-chain is constructed on each stack. The conventional walking EBP-chaining technique can basically obtain a stack trace from its stack.

However, the simple walking EBP-chaining technique fails in the first stage. WOW64_CONTEXT is saved when a system call is invoked. Thus, the first entry of an actual call-stack is usually a system call stub. System call stubs are typical cases for a function without using a frame pointer, as mentioned in Subsect. 2.2 A conventional technique at least overlooks a return address to the caller function of the system call.

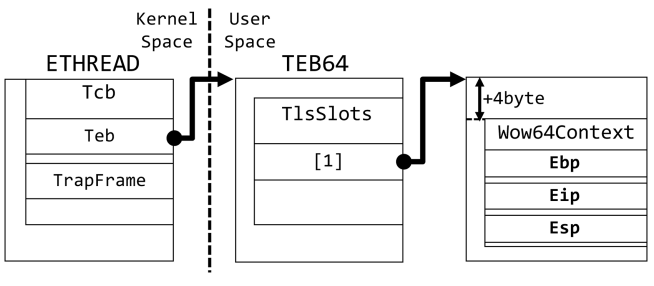

Fig. 6. Wow64Context held by WOW64 layer's thread local storages.

Sub-method (b-2) has a custom-tailored feature for system call stubs in the WOW64 layer. When a system call is invoked by a 32-bit application, there are two return addresses on the top of the stack pointed to by the ESP register saved in the WOW64_CONTEXT structure. The first one points inside the system call stub and the other points inside the caller to the stub. If the EIP pointing to the system call stub is confirmed, sub-method (b-2) obtains the second entry of the stack as a return address to the caller of the stub. After that, it starts walking EBP-chaining to the rest of the return addresses.

Evaluation

In this section, we confirm that our stack-trace method can accurately obtain stack traces of ordinary processes in a memory dump of Windows x64. We compared our Rekall plugin, which executes our method, with WinDbg to confirm the accuracy of our method. We also present the evaluation results.

Evaluation method

In this evaluation, we focused on processes originating from Windows official executable files, such as notepad.exe, calc.exe, and explorer.exe. WinDbg is Microsoft's official debugger and can also be fully supported by symbols provided by Microsoft symbol server service. Therefore, WinDbg definitely obtains the correct stack traces of such processes. We confirm that our plugin can obtain the same results as WinDbg's. We used a memory dump obtained from a virtual machine (VM) running on the KVM virtual machine monitor. The VM has 8-gigabyte memory, and Windows 7 x64 SP1 is installed on the VM.

x64 process

We first confirmed the effectiveness of our method for a x64 process in a memory dump of the Windows x64 environment. In this evaluation, we examined notepad.exe running on the environment when the dump had been obtained.

Fig. 7 shows two stack-trace results for a thread executing the main routine of notepad.exe. The upper half of the figure shows the results with the `k` command of WinDbg. The lower half shows the WinDbg-like results with our plugin. WinDbg outputs combined stack traces from both the kernel and user stack. In this evaluation, we ignored the stack trace from the kernel stack because our target was user applications.

Each row in Fig. 7 represents a stack frame for each nested function call. The Child-SP column represents the RSP value used by the row's function. The RetAddr column represents the return address used when the function is returned. The Call Site column represents the function name with offset. The column of our plugin seems to be different from WinDbg's. This is due to the symbol-mapping algorithm between Rekall and WinDbg. We can ignore this column because a virtual address indicated by this column equals the RetAddr entry on the previous row.

The debug messages of our plugin indicate that .pdata section of user32.dll is not present. Our plugin used sub-method (a-2-y) to obtain ⓒ and ⓓ rows. The other rows were obtained using sub-method (a-2-x).

For the reasons stated above, we compared the RetAddr column of entries marked ⓪ to ⑥ with those marked as ⓐ to ⓖ, respectively. We also compared Child-SP column entries ① to ⑥ with those of ⓑ to ⓖ, respectively. All entries of WinDbg and our plugin were equal.

Fig. 7. Stack-trace results with WinDbg (upper half) and our method (lower half).

In addition, both first entries ⓪ and ⓐ contain TrapFrame. Rekall abbreviates the most significant 16 bits of a virtual address for the x64 architecture because it is always 0xffff or 0x0000 in the current implementations of the x64 processor. Therefore, our plugin obtained a user context from TrapFrame at 0xfffff880023eec20, which was the same address as that with WinDbg.

We confirmed that our method could correctly obtain the stack trace of the x64 process running on Windows x64.

x64 process without metadata

We evaluated the effectiveness of our method for the x64 process without symbols and metadata. In this evaluation, to imitate a situation of obtaining a stack trace from code regions without metadata, both WinDbg and our plugin attempted to obtain a stack trace with no dependence on symbols and metadata. We examined notepad.exe, as mentioned Subsect. 5.2. This evaluation differed from that discussed in Subsect. 5.2 in that both WinDbg and our plugin did not use symbols on the user space. Specifically, WinDbg unloaded symbols of user32.dll and notepad.exe before executing the `k` command. Our plugin forcibly used sub-method (a-2-y).

Fig. 8 shows the result of WinDbg for this evaluation. WinDbg obtained only the first two entries of the correct stack trace shown in Fig. 7. However, our plugin obtained the same result shown in the lower half of Fig. 7. Therefore, we confirmed that our method could obtain the correct result without unwind information.

WOW64 process

In this evaluation, we confirmed the effectiveness of our method for the WOW64 process. We examined calc.exe running on WOW64 layer. Fig. 9 shows two stack-trace results for the main thread of the calc.exe. The upper half of the figure shows the result with WinDbg. The lower half shows the result with our plugin. The ChildEBP column represents the EBP value used by the row's function. The meaning of the RetAddr column is equivalent to that mentioned in Subsect. 5.2.

We compared each row's ChildEBP and RetAddr columns in the WinDbg result with those in our plugin result. All entries of WinDbg and our plugin were equal. We confirmed that our method could correctly obtain the stack trace of a 32-bit application on the WOW64 layer on Windows x64.

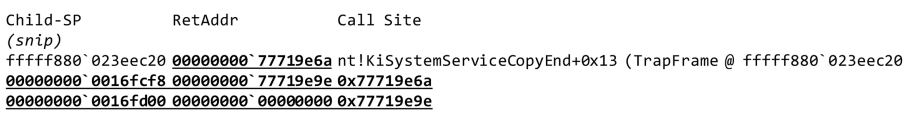

Fig. 8. Stack-trace result with WinDbg without using symbols on user space.

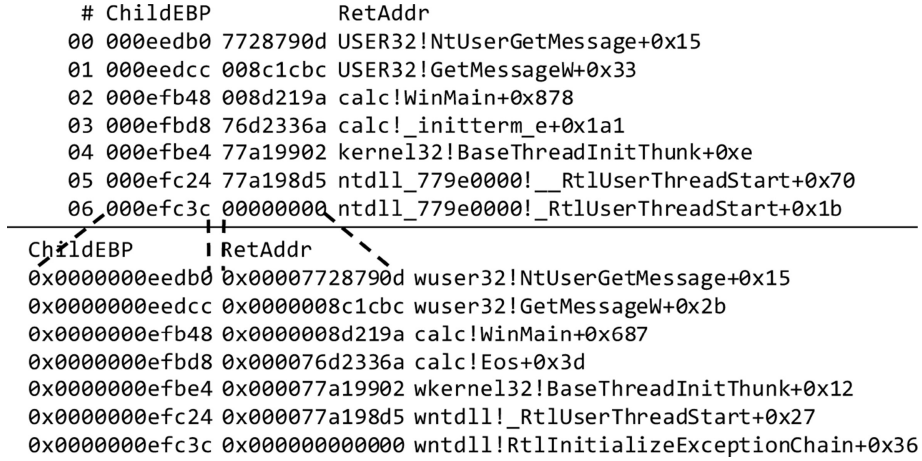

Fig. 9. Stack-trace results for WOW64-version calc.exe from WinDbg (upper half) and our method (lower half).

```
SP              RetAddr
0x000007a4f708: 0x07fefd9b1430 (kernelbase!RtlAnsiStringToUnicodeString+0x34), (snip)
0x000007a4f728: 0x07fefe6cd8a2 (ole32!DcomChannelSetHResult+0x3066), (snip)
0x000007a4f7d8: 0x07fefe6c93fa (ole32!CoSetState+0x35a), (snip)
0x000007a4f808: 0x0000776016e3 (kernel32!WaitForMultipleObjectsExImplementation+0xb3), (snip)
0x000007a4f868: 0x07fefeb34981 (shell32!SHCreateDirectoryExW+0xe99), (snip)
0x000007a4f898: 0x000077718f7d (user32!RealMsgWaitForMultipleObjectsEx+0xfd), (snip)
0x000007a4f908: 0x07fefeb34981 (shell32!SHCreateDirectoryExW+0xe99), (snip)
0x000007a4f938: 0x0000777162b2 (user32!MsgWaitForMultipleObjectsEx+0x2e), (snip)
0x000007a4f968: 0x07fefeafe6ca (shell32!SHGetSetSettings+0x1536), (snip)
0x000007a4f978: 0x0000777162e0 (user32!MsgWaitForMultipleObjects+0x20), (snip)
0x000007a4f998: 0x07fefd9b133c (kernelbase!SetEvent+0xc), (snip)
0x000007a4f9b8: 0x07fefeb32ad9 (shell32!SHGetPropertyStoreForWindow+0x1459), (snip)
0x000007a4f9c8: 0x07fefeb2f9ff (shell32!ILIsEqual+0x16f), (snip)
0x000007a4f9f8: 0x07fefe6c9b49 (ole32!CoSetState+0xaa9), (snip)
0x000007a4fa38: 0x07fefeb32d70 (shell32!SHGetPropertyStoreForWindow+0x16f0), (snip)
0x000007a4fab8: 0x07fefeb32cd0 (shell32!SHGetPropertyStoreForWindow+0x1650), (snip)
0x000007a4fae8: 0x07fefe6c2c1d (ole32!CoInitializeEx+0x1ed), (snip)
0x000007a4fb58: 0x07fefeb32de2 (shell32!SHGetPropertyStoreForWindow+0x1762), (snip)
0x000007a4fb88: 0x07fefe033843 (shlwapi!IUnknown_GetWindow+0x68f), (snip)
0x000007a4fbb8: 0x00007782004b (ntdll!RtlpTpWorkCallback+0x16b), (snip)
0x000007a4fc48: 0x000077820012 (ntdll!RtlpTpWorkCallback+0x132), (snip)
0x000007a4fc98: 0x00007781fc62 (ntdll!TpPostTask+0x2e2), (snip)
0x000007a4ff28: 0x0000775f59cd (kernel32!BaseThreadInitThunk+0xd), (snip)
0x000007a4ff58: 0x00007782a561 (ntdll!RtlUserThreadStart+0x21), (snip)
```

Fig. 10. Result of stack scanning for return address.

Comparison with conventional scan-based technique

We discuss the results of comparing our method and only using a conventional scan-based technique. Fig. 10 shows the result from the conventional scan-based technique for a thread of explorer.exe. Compared to the result from WinDbg for that thread, WinDbg's result did not include the equivalent of underlined entries shown in Fig. 10. Out of all 24 entries of the result, 10 were false positives with the scan-based technique. It is assumed that the root cause of such misdetection comes from the x64 Software Conventions. A function under the conventions allocate the maximum size of its necessary space in the stack. Such space holds its previous values before being used again. Therefore, return addresses pushed before then are easy to keep inside the active stack range.

We obtained two results with our plugin to confirm its accuracy. One was obtained using both sub-methods (a-2-x) and (a-2-y). The other was obtained using only sub-method (a-2-y). Both results are the same as that with WinDbg. Therefore, we confirmed that our method can obtain a stack trace more accurately than with only a conventional scan-based technique for a memory dump of Windows x64.

Case studies

In this section, we present case studies of real malware. In Sect. 5, we argued that our method can accurately obtain stack traces of ordinary processes. In this section, we discuss the effectiveness of our method for real malware.

Dridex

We first discuss the “Dridex” case, which is categorized as a banking trojan, whose main feature is stealing accounts of online banks.

We had rundll32.exe load the variant implemented as a DLL on a closed Windows 7 x64 SP1 environment and obtained memory dumps multiple times. We tried to obtain stack traces with our plugin and WinDbg for this sample. This variant provided many interesting cases for our method. In this subsection, we discuss three types of code regions that appeared while Dridex was running.

The first one is the original code region. The sample extracted its original code on its space. This behavior is called self-modification. The .pdata section was also overwritten and the number of RUNTIME_FUNCTIONs increased from 9 to 789, which implied that the unwind information was also restored when the original code was extracted. It seems that correct stack traces could be obtained by interrupting unwind information on memory in this case. However, both WinDbg and our plugin recognized some of that data as corrupted. Both may fail to correctly obtain stack traces of this Dridex. In Subsect. 7.2, we also discuss related but more malicious cases.

The second type is the injected region. The malware injected its DLL into other benign processes such as explorer.exe and iexplore.exe. Windows did not recognize the injected DLL regions inside the infected processes as image-mapped regions. For this reason, WinDbg could not recognize the regions as DLL. It merely outputted each value on the stack without interpreting the unwind information. However, our plugin could obtain stack trace without much problem because it checked the PE header on those regions regardless of Windows view.

The third type has MZ and PE signatures but no valid PE header. This header was completely corrupted. It almost invariably had only signatures. WinDbg failed to properly obtain stack traces for the same reason regarding the second type. This result was almost sequentially outputted each value on the stack. Our plugin first attempted to interpret that invalid PE header but failed. It used sub-method (a-2-x) for the same thread, and more accurately obtained stack trace than WinDbg.

Zeus P2P

This subsection introduces another interesting case we found in memory dumps that included frozen “Zeus P2P”. This variant injected code into other running processes. We examined these infected processes by using our method and found that return addresses pointed to UMS-related APIs such as RtlpUmsPrimaryContextWrap. The UMS means user-mode scheduling mechanism provided by Windows x64. The UMS previously caused memory corruption vulnerability (Microsoft, 2012). Some variants of Zeus P2P are often used with rootkit. Perhaps this Zeus P2P uses this vulnerability to elevate privilege and install the rootkit.

The UMS often develops into a gap between the kernel's and user's views for threads. As its name suggests, the UMS supports thread scheduling in user mode without helping the kernel-mode scheduler. To correctly obtain an execution context and stack trace for the UMS thread, our method should be equipped with custom-tailored functions for the UMS scheduler.

Tinba

We also introduce a case in which stack-trace techniques themselves are not effective. We found injected code regions in several benign processes, such as iexplore.exe, when analyzing memory dumps for "Tinba". Nevertheless, the stack-trace results indicate that no threads executed those code regions. This sample set hooks for several APIs after it had injected its code. It does not hold any threads for waiting until any one of the API hooks is triggered, the injected code activates only to handle that. Therefore, it is only during such hook-handling periods that these code regions appear in call-stack. Since the stack-trace techniques are nothing more than obtaining the call-stack at the current execution state, it is difficult to effectively use the techniques for forensic analysis for malware that uses waiting methods without holding threads. Another approach is needed to solve this problem.

On the other hand, the stack-trace techniques were effective in other cases as we already mentioned because those malware samples hold threads waiting for events during executing malicious code. Therefore, the stack-trace techniques in forensic analysis is effective for analyzing malware that holds threads to wait for events. For example, bot-type malware and remote access trojans generally wait for commands from the command and control server.

Discussion

In this section, we discuss our method's limitations and how it can be disrupted. We also discuss its applicability to other platforms.

Limitations

There are two issues in obtaining contexts and identifying return addresses.

First, we discuss the issue of obtaining contexts. A thread often has several or nested user contexts, such as UMS threads and callbacks from the kernel, in the Windows architecture. Sub-methods (a-1) and (b-1) obtain only the last context of a target thread. The current version of our plugin cannot correctly obtain an entire stack trace of a thread using such features. This issue can be addressed by also obtaining contexts saved by mechanisms providing the features.

The other issue is of identifying return addresses. In some cases, sub-method (a-2-y) cannot narrow the candidates to only one. Our method may find several candidates that have reachable paths to the current RIP. The current implementation simply selects the nearest candidate from the current RSP. In addition, as stated in Subsect. 4.3, our method cannot obtain a target of an indirect call via a register. Such a candidate is selected only if other candidates have no reachable paths. To address this issue, we are considering a deeper analysis for stack and code.

Disrupting our methods

We now discuss two disrupting methods against our method and their countermeasures.

One method involves embedding corrupted or completely fake unwind information into a PE file. Sub-method (a-2-x) and WinDbg are confused by such fake information. If malware pushes fake return addresses into the position in its stack specified by fake unwind information, both obtain completely false stack traces. Countermeasures against this problem involve verifying the results of sub-method (a-2-x) according to those of sub-method (a-2-y).

The other disrupting method is return-oriented programming (ROP) (Roemer et al., 2012). Stack usage of ROP is completely far removed from general programming techniques. With this technique, return addresses in a stack point to the next executing "gadgets", which mean small instruction sequences end up with a `ret` instruction. In the first place, stack-trace techniques are established upon the assumption that a return address is held on the stack to return the caller. Therefore, no stack-trace technique can obtain a call-stack from a stack used with ROP. A solution to this problem is to use the existing ROP analysis techniques (Graziano et al., 2016; Stancill et al., 2013).

Applicability to other platforms

We now discuss our method's applicability to other platforms. In this paper, we focused on solving issues of building stack traces from only memory dump of the Windows x64. However, the basic concepts can be applied to other platforms.

A common OS feature is that a user context is saved into memory by the OS when an event occurs. Therefore, user contexts can be obtained from memory dumps of other environments. In fact, there are already implementations targeting x64-version Linux (Smulders, 2013) and Windows x86 (Pulley, 2013; Ligh, 2011; Pshoul, 2017).

Building stack traces based on metadata can also be widely applied to other platforms. For example, the ELF image for the x64 architecture has metadata for exception handling as the .eh_frame section (Hubička et al., 2013; Hubička, 2003). If not being stripped, the ELF image itself contains symbol information. These metadata are useful for stack traces from a memory dump. In addition, flow-based verification can improve the accuracy of building stack traces for general functions with FPO optimization. More accurately stack walking can be enabled by combining the verification method with a conventional scan-based technique.

Related work

We now present related work on building stack traces from memory dumps.

There are four methods of stack analysis for Windows x86 memory. The first one (Arasteh and Debbabi, 2007) is a combination of stack and code analyses to obtain execution history. Its goal is to extract a possible execution path by correlating stack residues and an execution model. The second method (Hejazi et al., 2009) includes a stack-analysis method for extracting sensitive information such as username and passwords. This method detects such information passed to specific APIs by stack analysis. The last two methods are implemented as `exportstack` plugin (Pulley, 2013) and `malthstack` plugin (Pshoul, 2017) for The Volatility Framework (The Volatility Foundation, 2017). The plugins provide several options for stack analysis including building stack traces.

All methods use EBP-chaining-based and/or scan-based techniques to obtain stack traces. The first two methods find the location of a user stack by analyzing a kernel-side stack. The `exportstack` and `malthstack` plugins obtain the EBP register's value from ETHREAD.Tcb.TrapFrame in contrast to the other two

methods. All methods cannot be used for Windows x64 because they are specialized for only Windows x86.

There also are stack-analysis plugins targeting Linux x64 (Smulders, 2013) for The Volatility Framework. These plugins extract the execution state for each thread from Linux x64 memory samples. They can obtain the registers as saved on the kernel stack and obtain detailed analysis and annotation of the execution stacks.

Conclusion

We introduced a method for building stack traces from a memory dump of the Windows x64 environment in which frame-pointer chains are basically not constructed. Our method obtains a user context of each thread from a memory dump and emulates stack unwinding according to metadata for exception handling. We also introduced a flow-based verification method for extracting the correct return address from all candidates which are detected as return address by a conventional scan-based technique.

We confirmed our method's basic functionalities. It could obtain stack traces of the x64 process with or without metadata and WOW64 process. We also compared our method and using only a conventional scan-based technique and confirmed that our method could obtain stack traces more accurately than the other. We conducted experiments using real malware samples as case-studies and introduced interesting cases. We confirmed that our method's effectiveness for real malware such as bot-type malware. We also discussed our method's limitations, disrupting methods, and countermeasures against them. For future work, we will consider deeper stack analysis with sophisticated program-analysis techniques to restore or estimate an entire incident timeline.

References

Arasteh, A.R., Debbabi, M., 2007. Forensic memory analysis: from stack and code to execution history. Digit. Invest. 4, 114–125.

CodeMachine Inc, 2011. CodeMachine - Article - X64 Deep Dive. http://www.codemachine.com/article_x64deepdive.html.

Dolan-Gavitt, B., 2007. The vad tree: a process-eye view of physical memory. Digit. Invest. 4, 62–64.

Google Inc, 2017. Rekall memory forensic framework. http://www.rekall-forensic.com/.

Graziano, M., Balzarotti, D., Zidouemba, A., 2016. Ropmemu: a framework for the analysis of complex code-reuse attacks. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM, pp. 47–58.

Hejazi, S.M., Talhi, C., Debbabi, M., 2009. Extraction of forensically sensitive information from windows physical memory. Digit. Invest. 6, S121–S131.

Hubička, J., 2003. Porting gcc to the amd64 architecture. In: Proceedings of the GCC Developers Summit.

Hubička, J., Jaeger, A., Matz, M., Mitchell, M., Girkar, M., Lu, H., Kreitzer, D., Zakharin, V., 2013. System V Application Binary Interface Amd64 Architecture Processor Supplement (With Lp64 and Ilp32 Programming Models) Draft Version 0.3. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.

Johnson, K., 2007. How does one retrieve the 32-bit context of a wow64 program from a 64-bit process on windows server 2003 x64? http://www.nynaeve.net/?p=191.

Ligh, M.H., 2011. Mnin security blog: Investigating windows threads with volatility. http://mnin.blogspot.jp/2011/04/investigating-windows-threads-with.html.

Microsoft, 2017. Debugging with symbols (windows) - msdn - microsoft. https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588.aspx.

Microsoft, 2017. Wow64_context structure (windows) - msdn - microsoft. https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms681670.aspx.

Microsoft, 2017. Wdk and windbg downloads - windows hardware dev center. https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit.

Microsoft, 2017. x64 software conventions. https://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx.

Microsoft, 2017. Previousmode (windows drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff559860.aspx.

Microsoft, 2012. Microsoft security bulletin ms12-042-important. https://technet.microsoft.com/en-us/library/security/ms12-042.aspx.

Momot, F., 2013. Use of windows exception handling metadata. http://www.leviathansecurity.com/blog/use-of-windows-exception-handling-metadata.

Murakami, J., 2010, http://www.ffri.jp/assets/files/research/research_papers/psj10-murakami_EN.pdf. Exploring the x64. PacSec 2010.

OSR Open Systems Resources, Inc, 2009. Analyst's Perspective: X64 Trap Frames. http://www.osronline.com/article.cfm?article=542.

Pshoul, D., 2017. community/dimapshoul at master volatilityfoundation/community github. https://github.com/volatilityfoundation/community/tree/master/DimaPshoul.

Pulley, C., 2013. Github - carlpulley/volatility: A collection of volatility framework plugins. https://github.com/carlpulley/volatility.

Roemer, R., Buchanan, E., Shacham, H., Savage, S., 2012. Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. 15 (1), 2:1–2:34.

Smulders, E., 2013. Github - dutchy-/volatility-plugins: Container for assorted volatility plugins. https://github.com/Dutchy-/volatility-plugins.

Stancill, B., Snow, K.Z., Otterness, N., Monrose, F., Davi, L., Sadeghi, A.-R., 2013. Check my profile: leveraging static analysis for fast and accurate detection of rop gadgets. In: Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145. Springer-Verlag New York, Inc, pp. 62–81. RAID 2013.

The Volatility Foundation, 2017. The volatility foundation - open source memory forensics. http://www.volatilityfoundation.org/.