Mixed-Mode Malware and Its Analysis

Shabnam Aboughadareh, Christoph Csallner
University of Texas at Arlington
Arlington, TX 76019, USA
shabnam.aboughadareh@mavs.uta.edu,csallner@uta.edu

Mehdi Azarmi
Purdue University
West Lafayette, IN 47907, USA
mazarmi@purdue.edu

ABSTRACT

Mixed-mode malware contains user-mode and kernel-mode components that are interdependent. Such malware exhibits its main malicious payload only after it succeeds at corrupting the OS kernel. Such malware may further actively attack or subvert malware analysis components. Current malware analysis techniques are not effective against mixed-mode malware. To overcome the limitations of current techniques, we present an approach that combines whole-system analysis with outside-the-guest virtual machine introspection. We implement this approach in the SEMU tool for Windows. In our experiments SEMU could successfully analyze several mixed-mode malware samples that evade current analysis approaches. The runtime overhead of SEMU is in line with the most closely related dynamic analysis tools TEMU and Ether.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Performance, Security

Keywords

Dynamic malware analysis, mixed-mode malware, rootkit

1. INTRODUCTION

Malware analysis has been studied widely, using many real-world malware samples. However we are not aware of existing work that exposes malware analysis to *mixed-mode* malware. Mixed-mode malware is a type of malware that (a) has interdependent user- and kernel-mode components and (b) may actively attack or subvert malware analysis components. We say that malware components are interdependent if the second component performs its main malicious payload only if the kernel manipulation of the first component succeeds.

Malware poses significant challenges to modern society. Among others, malware can take control of a victim system and perform arbitrary actions. For example, malware can log individual keystrokes to steal online banking passwords and change OS kernel data structures to remain undetected for years. These challenges are real and affect many people. For example, a survey conducted in January 2011 found that *one third* of the 2,089 U.S. online households surveyed had been victims of malware in the previous year [8]. The survey estimated that in the previous year malware cost U.S. consumers overall USD 2.3 billion.

Given the big impact malware has, it is important for malware analysts to analyze malware and develop countermeasures. For such malware analysis, an important technique is to monitor the execution of actual malware with state-of-the-art dynamic malware analysis tools such as those based on TEMU [31], Anubis [5], and Ether [9]. Monitoring malware executions allows malware analysts to reverse-engineer and understand the subtle details of how a concrete malware instance functions. Analysts can leverage such understanding when designing and deploying malware countermeasures. Such countermeasures can ultimately protect a wide range of computers from both the specific malware analyzed and from similar, derived classes of malware [5].

Current dynamic malware analyses [31, 9, 5, 34, 16] are not effective for analyzing mixed-mode malware. The reason is that existing malware analysis tools suffer from one or both of the following shortcomings.

(1) First, many current analysis techniques place analysis components in the domain in which the malware is executing and thereby expose the analysis to malware manipulations. These approaches are referred to as *inside-the-box*, *inside-the-guest*, on *in-guest*. For example, popular analysis platforms such as TEMU and Anubis run the malware in a virtual machine. To inspect the state of the malware and the VM, such malware analyses often place some virtual machine introspection (VMI) components inside the VM, which exposes VMI and thereby the entire analysis to malware manipulation.

(2) Second, many approaches focus on a single domain, either kernel-mode or user-mode, but fail to fully capture malware that operates in both modes [9, 35]. For example, Ether leverages hardware virtualization extension to operate outside-the-guest but focuses only on user-mode analysis.

PPREW-4 December 09 2014, New Orleans, LA, USA

Ether relies on the integrity of the kernel when inspecting the system state and malware behavior. However, mixed-mode malware manipulates the OS kernel and thereby foils such single-mode analysis.

To address the limitations of the existing techniques and analyze mixed-mode malware effectively, we propose a novel dynamic malware analysis tool called SEMU (*Secure EMUlator*). SEMU operates both outside-the-guest and across kernel and user modes. In our experiments we also found that SEMU's overhead was in line with closely related existing tools, i.e., Ether and TEMU. While our current SEMU implementation is for Windows, our analysis approach could also be implemented for other operating systems such as Linux. To summarize, this paper makes the following major contributions.

- We describe and provide practical implementations of several mixed-mode malware samples. Mixed-mode malware cannot be fully analyzed with current state-of-the-art dynamic malware analysis tools such as those built on TEMU, Anubis, and Ether.
- We present SEMU, a whole-system outside-the-guest dynamic malware analysis tool that can effectively analyze mixed-mode malware. For example, SEMU *detects* and *analyzes* kernel exploits that cannot be analyzed by current kernel-mode analysis approaches.
- We provide the first empirical evaluation of the runtime characteristics of a whole-system outside-the-guest dynamic malware analysis tool such as SEMU.

2. BACKGROUND AND RELATED WORK

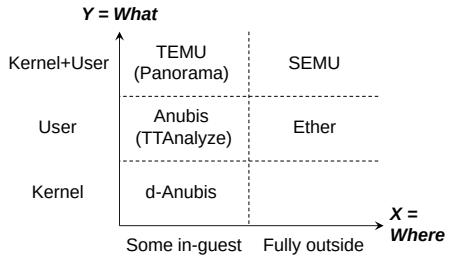

Figure 1: Two dimensions of dynamic malware analysis: Scope (y-axis) and whether an analysis places some of its components inside-the-guest (x-axis).

We can classify dynamic malware analyses along two dimensions. Figure 1 shows on the x-axis whether an analysis places some of its components inside the guest OS. Having components inside-the-guest makes an analysis vulnerable to malware attacks. The y-axis captures the scope of the malware analysis. SEMU is the only analysis that combines a kernel+user scope with being fully outside-the-guest.

2.1 Virtual Machine and Introspection

Malware may corrupt the OS it is running on and may in turn corrupt other programs running on the OS, including malware analyses. To retain control of the machine, malware is thus typically run on a (guest) OS that is installed on a hardware emulator or *virtual machine* (VM) [4, 31, 9]. The VM runs on another OS called the host OS. A VM also enables an analyst to inspect all interactions between malware and guest OS.

To be useful, a VM-based malware analysis tool has to query the current VM state. This state is readily available in a low-level form, i.e., in terms of register values and main memory bytes. But a malware analysis tool is typically written in terms of higher-level OS abstractions such as threads and processes. This gap between the readily available low-level hardware state and the desired high-level operating system state is called the *semantic gap* [10].

Virtual machine introspection (VMI) bridges the semantic gap, by reconstructing high-level OS information from low-level hardware state [14]. Without bridging the semantic gap, even basic analysis tasks such as logging the execution trace of a malware sample become very hard if not impossible.

2.2 Inside- vs. Outside-the-Guest VMI

Since malware may corrupt anything within its reach, it is useful to determine if a VM-based malware analysis places any of its components inside the guest OS. This classification notably differs from the more common classification of where the majority of the malware analysis components reside. While TEMU and Anubis are often described as outside-the-guest [2], they place at least some of their VMI components inside-the-guest.

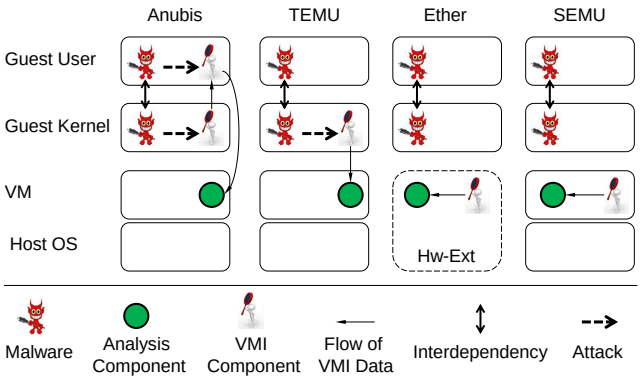

Figure 2: Architecture comparison. Not shown is malware's main payload attacking the guest OS. Hw-Ext = Hardware virtualization extension.

Figure 2 gives an overview of state-of-the-art tools including TEMU [31] and Anubis [5] that both place a custom kernel-mode VMI driver inside the guest OS. Such a driver is outside the malware's reach if the malware is restricted to user-mode. The existing drivers differ in how they pass OS state to the analysis. The TEMU driver writes to a predefined I/O port. The Anubis driver reports to a user-mode application, which communicates with the analysis component over a virtual network. The Anubis extension dAnubis [23] patches the kernel functions that load kernel-mode modules. The patched functions then notify dAnubis when kernel-mode malware is loaded.

Ether [9] performs outside-the-guest VMI by relying on a processor-specific hardware virtualization extension (Intel VT-x [7]). The hardware extension allows Ether to run in a privileged hypervisor mode. Certain events in the monitored VM such as some exceptions trigger the hypervisor mode and return control to Ether. For example, to log an

instruction, Ether sets a trap flag before the instruction to force a debug exception that returns control to Ether.

While outside-the-guest VMI protects Ether from some malware attacks, relying on hardware virtualization extensions has two drawbacks. First, for a fine-grained analysis that logs every instruction, Ether has to install a trap flag before every instruction and handle the resulting debug exceptions. This single-step mode slows down execution by three orders of magnitude [36]. Second, Ether requires that the underlying processor supports hardware virtualization extensions, but many processors do not meet this requirement.

Besides well-known analysis tools we described, there are several VM-based approaches that focus on malware and rootkit detection or protection [6, 30, 38, 25, 33, 18, 17, 29, 32]. However none of these existing tool can *monitor* and *log* the execution of kind of mixed-mode malware or kernel exploits described in this paper.

2.3 Scope: Single-Domain vs. Whole-System

Many analyses capture either user-mode or kernel-mode malware activities, but not both. Such a single-mode focus is insufficient for fully analyzing mixed-mode malware.

An example whole-system analysis is Panorama. It is a part of BitBlaze [31] and uses TEMU for performing whole-system taint analysis [39]. But as TEMU, Panorama performs some VMI tasks inside the guest OS. Panorama and similar approaches track the information flow of sensitive data such as keystrokes and network packets to detect and analyze malware [21, 26, 22, 37, 20]. But mixed-mode malware such as Figure 3 may not have such data flows and thus cannot be fully analyzed by such approaches.

An example user-only analysis is Ether. It assumes the integrity of kernel data (i.e., the system call table) and control-flow. A mixed- mode malware can drop a kernel-mode malware that manipulates the system call table and mislead Ether's analysis.

An example kernel-only analysis is the Anubis extension dAnubis [23]. It is notified whenever malware is loaded into kernel memory. But being single-mode, dAnubis may miss attacks from user-space. For example, user-mode malware can access the kernel via zero-day exploits, such as bugs in standard device drivers. In such cases a malicious payload executes with kernel privileges without loading a kernel-mode module.

3. MIXED-MODE MALWARE

At a high level, mixed-mode malware runs in two phases. (1) In phase one, a (typically kernel-mode) malware component modifies a part of the OS kernel, i.e., kernel code, kernel data, or both. (2) In phase two, a (typically user-mode) malware component executes the main malicious payload. There are three key ideas behind these phases.

The first key idea is that the payload reads modified kernel data or invokes modified kernel code. The semantics of the executed payload is thus determined by the success of the phase 1 kernel manipulation attempt.

The second key idea is that a malware analysis can only observe the main malicious payload behavior if the phase 1 kernel modification attempt succeeds. If the phase 1 attempt did not succeed, then the phase-2 payload may amount to benign behavior or a different malicious behavior.

The third key idea is that the phase-1 kernel modification *may* also lead current malware analysis tools to incomplete or inaccurate analysis results. Mixed-mode malware thereby effectively prevents malware analysis with current tools.

As other malware, mixed-mode malware has various implementation options. Phase-1 can be carried out by a kernel-mode component that has been deployed by either a user-mode component or by a user-mode dropper application. But it may also be triggered from a user-mode component via a zero-day kernel exploit which manipulates kernel structures. Similarly, phase 2 is typically carried out by a user-mode component, as it is more convenient for malware writers. But phase 2 could also be implemented by a kernel-mode component.

More formally, the state of the kernel at time T consists of code C_T and data D_T. Phase 1 performs operation M to manipulate the original kernel code C_O and data D_O, yielding C_N and D_N, or $M(C_O, D_O) = (C_N, D_N)$. If the manipulation succeeds then C_N is the manipulated code C_M and D_N is the manipulated data D_M. In the new system state, phase 2 executes a sequence of operations U, which invoke C_N and read D_N. The main malicious payload behavior, operation P, is the sequence of commands executed with successfully manipulated kernel data and code, $U(C_M, D_M)$.

$$U(C_N, D_N) = \begin{cases} \text{operation P} & \text{if } C_N = C_M \wedge D_N = D_M \\ \text{operation X} & \text{otherwise; } X \neq P \end{cases}$$

Optionally, operation M may also attack an inside-the-guest VMI component of a malware analysis A. In this case the analysis produces an incomplete, incorrect, or misleading report if M succeeded. Since the malicious payload is only observable if M succeeds, analysis A is not effective for analyzing such mixed-mode malware.

In the following we describe concrete mixed-mode malware examples that cannot be analyzed by current malware analyses. We present these examples in a minimal style for ease of exposition. The examples are minimal in the sense that each example exploits one weakness of one kind of current malware analysis technique. However these examples could be combined into integrated, comprehensive malware attacks that cannot be fully analyzed by several or all current malware analysis techniques.

3.1 Misleading User-Only Analysis

This section describes how mixed-mode malware can mislead user-only analysis, regardless of where the analysis performs VMI. These techniques thus affect both in-guest VMI such as Anubis and outside-the-guest VMI such as Ether. The high-level idea is to modify those guest OS kernel entities in phase 1 that the phase 2 malicious payload uses. Since kernel modifications are outside the scope of a user-focused analysis, the analysis misses the true semantics of the malicious payload.

Figure 3 shows the main steps of misleading user-only analyses such as Anubis or Ether. (1) First the user-mode malware Mal.exe installs a rootkit (a kernel-mode component). The rootkit creates a fake system call table that contains pointers to itself instead of to the standard operating system services. (2) Then the rootkit overwrites the pointer (within the `KTHREAD` object of Mal.exe) the operating system uses to find the address of the system call table with the address of the fake system call table.

(3) Each time Mal.exe calls system call A the operating system will now (4) follow the pointer to the fake system call table. (5) The OS thereby directs the control-flow to the

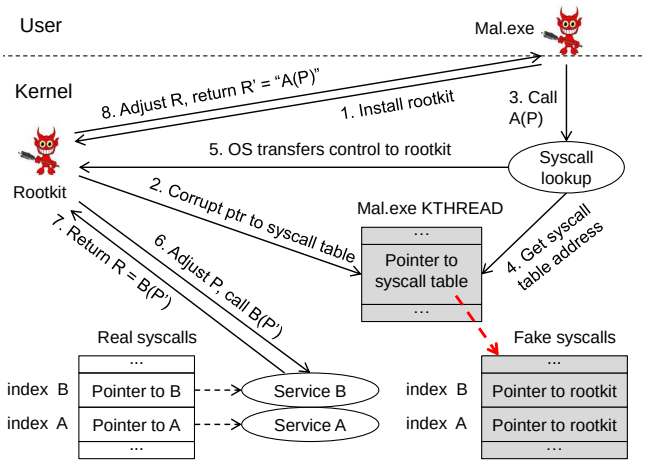

Figure 3: Example malware misleading Ether's user-only VMI; box = data; oval = function; solid arrow = attack; dashed arrow = original pointer; bold dashed arrow = manipulated pointer; gray = corrupted; P = system call A parameter values; P' = system call B parameter values.

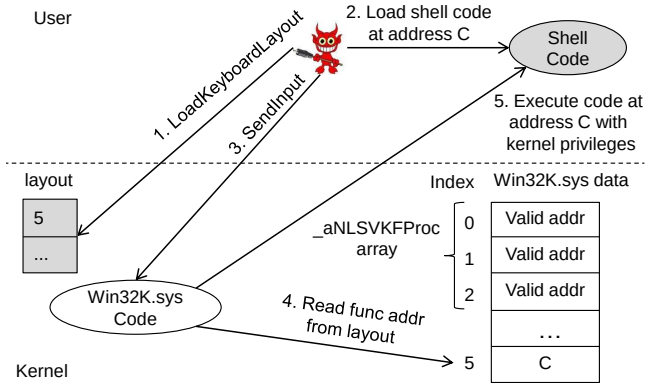

Figure 4: Stuxnet kernel exploit; box = data; oval = function; white = original; gray = corrupted.

rootkit instead of the original OS service. (6) The rootkit adjusts the input parameters to system call B and invokes the corresponding OS services. (7) After executing the invoked system service, the rootkit adjusts the return values to the original system call and (8) returns the results to the user-mode malware component.

Since a user-only analysis logs system calls at the interface between user- and kernel-mode, it will log the system calls as they are issued by Mal.exe. If the kernel modification succeeds, the analysis will log a sequence of system calls that differs from the actually executed system calls.

If the kernel modification does not succeed then an analysis tool cannot observe the main malicious payload, as the system call table will not point to the malware-created one. The commands executed by Mal.exe thus lead to system calls that differ from the malicious payload.

3.2 Misleading Kernel-Only Analysis

This section describes how a mixed-mode malware sample cannot be fully analyzed by a kernel-only analysis, regardless of where it performs VMI. The malware sample thus affects both in-guest and outside-the-guest VMI. We are not aware of any current outside-the-guest VMI kernel-only analysis systems. But dAnubis is a well-known in-guest VMI kernel-only system [23].

The Stuxnet example malware does not load any malicious kernel-level code and is thus not tracked by a kernel-level analysis tool such as dAnubis. Instead, Stuxnet runs in user-mode. To obtain administrator access to the victim system, it executes some instructions (shellcode) with kernel privilege. This attack uses a vulnerability in the Win32k.sys driver of Windows XP and Windows 2000. Win32K.sys is the driver that manages the graphical user interface environment, e.g., by dispatching keyboard and mouse inputs to applications. A Windows user-mode application can create a custom keyboard layout file and activate it via the LoadKeyboardLayout API.

To hijack the kernel execution, Stuxnet passes a malformed keyboard layout file as an input argument to a system call. Stuxnet exploits that the kernel does not check the bounds of an array index that is provided in an user-generated keyboard layout file.

For finding the virtual key that corresponds to a keystroke, Win32k.sys obtains an index into an array from the active keyboard layout file. Win32k.sys gets the address at the given index and executes the function at this address. The attack crafts a keyboard layout file that contains an out-of-bounds index such as 5, which points to C. When interpreted as a function pointer, C points within the range of the malware's user-mode address space, which effectively yields control to the malware.

Figure 4 summarizes the main steps. (1) The user-mode malware loads a crafted keyboard layout file by invoking LoadKeyboardLayout. (2) Then it allocates memory at address C and loads a malicious shellcode into this address. (3) To trigger the vulnerability, it invokes the SendInput API, which synthesizes keystrokes and triggers Win32k.sys to use the crafted layout file for extracting the virtual key for the requested keystrokes. (4) Therefore Win32k.sys reads the value C as the function address and (5) executes the malicious shellcode at location C with kernel privilege.

To summarize, if the kernel modification succeeds the malware executes code at C with kernel privileges, untracked by dAnubis. If the kernel modification does not succeed to provide administrator access for malware code, malware cannot execute its malicious payload.

3.3 Misleading Inside-the-guest VMI

This section describes how mixed-mode malware can defeat inside-the-guest VMI, regardless of the scope of the malware analysis. This approach can thus mislead user-only techniques such as Anubis, kernel-only techniques such as dAnubis, and whole-system techniques such as those based on TEMU.

Figure 5 shows an example malware that can mislead in-guest VMI techniques such as TEMU's Module Notifier VMI driver. In Figure 5 the main malicious payload is to call service A with certain parameters (step 3). The malware Mal.exe triggers this payload only if the previous kernel manipulation attempt in step 2 succeeds. But step 2 also breaks TEMU's VMI scheme and thereby renders TEMU ineffective.

An analyst therefore has two options. She can allow step 2

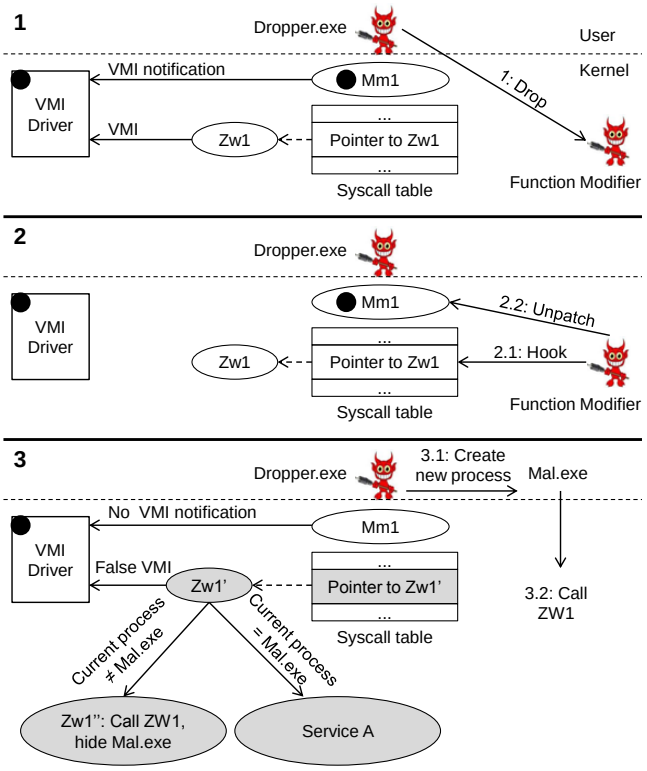

Figure 5: Example malware payload (calling A) that misleads TEMU; box = data; oval = function; dashed arrow = pointer; black dot = VMI entity; gray = corrupted.

to trigger the malicious payload, but this renders TEMU ineffective. The other option is to prevent the step 2 manipulation, but this prevents the main malicious payload from executing.

In more detail, Dropper.exe first drops the "Function Modifier" rootkit (step 1). This rootkit manipulates TEMU's VMI infrastructure (step 2). The rootkit checks the system call table and internal OS functions and unhooks and unpatches the functions TEMU uses. Then the rootkit manipulates these functions so they return false results to TEMU's VMI driver.

Specifically, the rootkit unhooks the internal OS function `MmLoadSystemImage` (Mm1), which TEMU hooks to detect driver loading by malware. The rootkit also unhooks the OS function `ZwQuerySystemInformation` (Zw1) of the system call table, which TEMU uses to retrieve driver module information. The rootkit then hooks Zw1 to point to an alternative malicious implementation Zw1'.

In step 3, the Dropper creates a new process for Mal.exe, which calls Zw1. If the phase-1 manipulation of step 2 does not succeed, then this results in a notification to TEMU, but no call to service A. If step 2 succeeds, then this call by Mal.exe results in the malicious payload and misleading TEMU's VMI.

3.4 Analysis Requirements

From the threat model and example mixed-mode malware attacks we can infer the following three requirements for effective analysis of mixed-mode malware. (1) The analysis must run outside the malware scope. For mixed-mode malware this means that the malware analysis cannot have any component such as VMI in the guest OS. (2) A precise and comprehensive model of both kernel data and kernel code. This is a common malware analysis requirement. Without such a model an analysis tool cannot generate a precise and comprehensive log of potentially malicious activities. (3) Log potentially malicious activities regardless of where they occur. This means monitoring and logging both kernel-mode and user-mode activities. Such logging requires identifying which instructions in user- and kernel-mode are run on behalf a malware component, for example, when the OS performs a requested system call.

4. SEMU DESIGN

This section describes SEMU's high-level design concepts. SEMU follows earlier approaches such as TEMU in that it uses virtualization. However SEMU places all analysis components outside the guest OS and its analysis covers both user-mode and kernel-mode malware codes. SEMU's concepts can be implemented in various ways. To evaluate SEMU we implemented it on top of QEMU. However the concepts are general and could be re-implemented using other virtualization techniques, such as various software-based virtual machines or via hardware virtualization extensions.

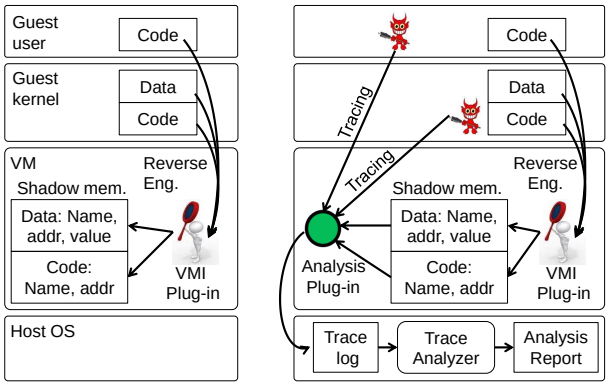

Figure 6: SEMU architecture and main execution phases: Pre-malware-execution phase (left) and the malware execution and post-execution log-analysis phases (right).

Figure 6 gives a high-level overview of SEMU's architecture and main execution phases. SEMU consists of the following four major components. (1) First, SEMU has a reverse-engineered model of the guest OS. (2) Second, SEMU's pre-execution phase copies key OS level elements from the guest OS into SEMU's *shadow memory* (Figure 6 left). The right part of Figure 6 shows the remaining components. (3) Malware execution and monitoring keeps SEMU's shadow memory in sync with the guest OS and creates a precise log of malware activities. (4) The last component is a post-execution log-analysis.

SEMU logs and keeps track of both user and kernel events. These events include system calls, I/O control (IOCTL), calls to functions exported by dynamic-link libraries (DLLs), and kernel-mode function calls. Such a comprehensive log captures information flow across the kernel-user divide, such

as user-mode code calling a system call and kernel code branching into user-space code.

Before the VM executes a guest instruction, SEMU decides whether to log the instruction. When making this decision, SEMU consults a precise and comprehensive memory model of the guest OS. This model is SEMU's shadow memory, which includes address ranges of code and data as well as individual data values.

SEMU builds and maintains its shadow memory from outside the guest OS and thus outside malware's reach. SEMU builds its initial shadow memory version by traversing the guest OS memory before malware execution. SEMU interprets the guest OS memory by consulting its *reverse-engineered model of the Windows OS* that includes symbol information (e.g., the layout of all kernel data structures) and the addresses of kernel functions. During analysis SEMU keeps its shadow memory updated by keeping track of events in the guest OS that change the kernel data. Such events include the creation and deletion of kernel objects and writes of kernel object fields.

4.1 Reverse-engineered OS Model: PDB

SEMU performs VMI by interpreting the guest OS memory from outside the guest. To parse the current guest OS memory, SEMU leverages the memory layout information provided in PDB (program database) files [28]. SEMU uses this approach because PDB data cover a wide range of kernel data structures in a wide range of Windows operating systems. While there has been promising recent work on synthesizing outside-the-guest VMI tools automatically [10, 12, 13], these synthesized VMI tools do not cover all the kernel data structures that are needed for malware analysis [10, Section 3A]. In future work we plan to explore integrating such synthesized VMI tools into our PDB-based approach.

To extract the precise behavior of the malware, SEMU needs to know where important member fields of OS objects and data structures are. For instance, to detect the Figure 3 malware attempt of changing the system call table, SEMU has to detect a manipulation of the system call table pointer within the `KTHREAD` object. SEMU thus tracks the manipulation of `KTHREAD` objects including field writes. These data structure layouts are documented as PDB symbols, together with the name and offset address of internal kernel-mode functions of the OS and drivers.

The format of data structures and other symbols differs across Windows versions. To build an accurate model, the VMI plug-in first resolves the guest OS and device driver version numbers. Then SEMU downloads the corresponding PDB symbols from Microsoft servers.

SEMU differs from the OS reverse engineering of current tools such as Volatility [1] and Virt-ICE [24]. Volatility is an off-line forensic analysis tool that does not monitor or log malware actions in kernel- and user-mode. Virt-ICE is an interactive debugger but does not monitor kernel manipulations and thus it is not effective against mixed-mode malware.

4.2 Pre-Execution: Create Shadow Memory

Directly before malware execution starts, SEMU initializes its shadow memory by copying guest OS code information and data into its shadow memory (Figure 6 left). SEMU performs this pre-execution phase before every execution of the malware, as the guest OS state may change between subsequent malware executions.

Algorithm 1: Main steps of the pre-execution phase.

```
1 On_init_event()
2 begin
3 |   trusted_code = phyAddr(kBase, kPE, drvBase, drvPE);
4 |   fMap = resolve(PDB, kBase, kPE, drvBase, drvPE);
5 |   dMap = resolve(kpcr_pointer);
6 |   current_proc = get_cr3_from_kprocess(kpcr_pointer);
7 end
```

Algorithm 1 summarizes the pre-execution phase. The algorithm basically initializes the following four key data structures. First, SEMU infers the address range of each trusted kernel code component and stores it in *trusted_code*. This includes ntoskrnl.exe, Win32k.sys, and other basic device drivers such as tcpip.sys and disk.sys. Subsequent phases use these address ranges to distinguish trusted from non-trusted kernel code. This step is important as monolithic operating systems such as Windows and Linux operate large amounts of code and drivers in kernel-mode, without address-space separation to isolate the kernel from possibly malicious code.

Second, SEMU creates *fMap*, a detailed list of functions within each trusted code component. For each function, fMap contains its name and its entry point address and name of the trusted code it belongs to.

Third, SEMU creates *dMap*, a detailed structure of key OS objects. dMap contains the name, address range and field values of many OS-level objects. Finally, SEMU stores the set of process objects also in *current_proc*.

SEMU retrieves these guest OS data by traversing the large number of OS objects that are reachable through the x86/x64 segmentation registers `FS` (x86), `GS` (x64). SEMU interprets the guest OS memory using its OS model reverse-engineered from PDB symbols. When in kernel-mode, the `FS/GS` register points to a kernel-mode data structure called kernel processor control region (`KPCR`). `KPCR` gives access to base addresses and PE[2] information of both kernel components (kBase, kPE) and drivers (drvBase, drvPE).

Via `KPCR`, SEMU retrieves information about both dynamic and static kernel addresses. A static address does not change during normal OS execution. Example static addresses include the interrupt table `IDT`, the system call table `SSDT`, and the global descriptor table `GDT`. Dynamic addresses may change during normal OS execution. Examples include OS process objects in the OS heap managed by the kernel's object manager such as `ETHREAD` and `EPROCESS` [28].

A special case of dynamic objects is the list of current processes current_proc. Via `KPCR`, SEMU extracts the base address of the page directory for each running process from the `KPROCESS` object's `DirectoryTableBase` field. Subsequent phases use this process list to track malware processes.

When executing an instruction in user-mode, the `FS/GS` register points to a thread environment block (`TEB`) user-mode data structure. A `TEB` contains information about the currently running thread and points to a process environment block (`PEB`). SEMU uses the `PEB` to resolve user-level

[1] http://code.google.com/p/volatility/

[2] http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

information about the current process.

PEB also indirectly points to the InloadOrderModuleList list of the loaded dynamic-link libraries (DLLs) within a process memory. SEMU identifies each of these DLLs by the LDR_MODULE, which includes information about its corresponding DLL such as the base (start) address of a DLL in the process address space. By using the base address of a DLL and extracting the offsets of its functions from the export table in the PE header, SEMU finds the entry point addresses of library functions in the process memory.

4.3 Whole-System Malware Analysis

When the malware executes, SEMU monitors and logs key events in both user- and kernel-mode. That is, unlike the user-only tracing common in current tools such as TEMU, Anubis, CWsandbox, and Ether, SEMU monitors and logs control flow in the kernel whenever the processor switches to the kernel to serve a malware's request. SEMU thereby discovers system call swapping attacks such as Figure 3.

To perform whole-system malware analysis the analysis plug-in distinguishes user-mode from kernel-mode malware code. Address-space separation in user-mode makes it easy to identify the user-mode malware instructions. But in kernel-mode we need to monitor and log instruction execution in the following two cases: (1) First, whenever the processor switches to kernel-mode to execute the request of user-mode malware (e.g., to perform a system call); and (2) Second, when an untrusted kernel instruction (injected kernel code or dropped driver module) executes.

Algorithm 2: Whole-system tracing of malware operations.

```
1  begin
2      if CS ∈ User and CR3 ∈ malware_proc_CR3 then
3          return trace_user;
4      end
5      if CS ∈ Kernel then
6          if phy_addr (current_instruction) ∉ trusted_code
           then
7              return trace_kernel;
8          else
9              kthread = current_user_thread (kpcr_pointer);
10             if kthread ∈ malware_proc then
11                 return trace_kernel;
12             end
13         end
14     end
15     return dont_trace;
16 end
```

Algorithm 2 summarizes how the analysis plug-in enables whole-system monitoring and logging. In line 2 the analysis plug-in checks the processor mode and the value of the CR3 register. (The CR3 value is the base address of the page directory for the currently running process.) If the processor performs an instruction from user space and the CR3 value belongs to a stored list of page directory base addresses of the malware process and new processes that the malware has created, the analysis plug-in enables user-mode tracing. Otherwise, if the processor works in kernel-mode, the analysis plug-in enables tracing if the current instruction is untrusted (line 6) or the current KTHREAD object belongs to a user-mode malware thread (line 10).

When tracing is enabled, SEMU provides two main logging options. The low-level logging option logs each instruction. The high-level option uses the data stored in the shadow memory to provide a high-level summary in terms of the names and addresses of both the invoked functions and the accessed data objects. In user-mode, the resulting log includes library calls, system calls, and IOCTLs. In kernel mode, the resulting log includes, besides others, the kernel code and data manipulation whenever an untrusted code attempts to modify a memory location that is included in the shadow memory. SEMU thereby detects and logs function pointer hooking attacks and kernel data manipulations such as DKOM (Direct Kernel Object Manipulation).

For the Figure 3 example, SEMU logs any malware system calls from user-mode in step 1, operations performed by the rootkit in steps 1 and 2, the malware's service A system call in step 3, and the rootkit's execution of service B in step 6.

During malware execution, SEMU keeps its shadow memory in sync with the guest OS. This is done by the VMI plug-in, which tracks the execution of kernel functions that load new code or create, modify, or delete objects. SEMU then reflects such operations in its shadow memory. The VMI plug-in updates the shadow memory by adding the addresses of newly created objects and removing the addresses of deleted objects from the dMap. For this purpose the VMI plug-in monitors changes in the OBJECT_DIRECTORY structure after execution of several OS functions that allocate and deallocate memory in the kernel, such as RtlAllocateHeap, ExAllocatePoolWithTag, ExFreePoolWithTag, and RtlFreeHeap. When malware overwrites trusted code, SEMU similarly removes the overwritten code range from trusted_code.

4.4 Malware Logging

	Description
E	EP addr+name, M name
C	C addr+name, T addr+name, T M name, M name
W	Inst addr, D addr+name, F name, M name
R	Inst addr, D addr+name, F name, M name

Table 1: Kernel execution trace format; EP = entry point; Addr = address; C = caller; T = target; M = module; D = data; F = member field; Inst = current instruction.

Table 1 shows the format of the trace file the analysis plug-in collects from the kernel during malware execution. SEMU creates this log with information from its shadow memory, such as addresses and names of functions and objects. **E** represents the execution of a function. Whenever a function executes, the analysis plug-in logs its entry point address and its name (if the VMI plug-in has resolved the name). Since there is no resolved name for the functions that execute within kernel-mode malware, the analysis plug-in only writes the address and the name of the malware's kernel-mode module or U (untrusted code) for dropped drivers and injected codes.

For each executed control transfer (**C**) to an address we have in our shadow memory, the analysis plug-in writes the caller's address and the module name as well as the target's address and the module name. The analysis plug-in traces accesses (**W**=write and **R**=read) within the address range of a kernel object as follows. SEMU logs all direct writes

of kernel data performed by untrusted code. SEMU logs the name of kernel data and its overwritten field members. SEMU also monitors writes of kernel data by the memory management functions that malware invokes.

The analysis plug-in also tags manipulated data structures. Whenever a read operation occurs within a manipulated kernel data structure, SEMU logs which kernel functions are affected by manipulated kernel data.

4.5 Post-Execution: Log Analysis

In the final step of mixed-mode analysis SEMU's trace analyzer produces a human readable report. The report contains name and address of modified kernel data as well as the internal OS functions that execute after user-mode requests and the functions that referred to manipulated OS objects.

SEMU contains a trace analyzer application that performs post-execution operations. The post-execution aggregates the collected log, for example, matching system calls from user-mode with operations invoked by kernel-mode malware. For instance, in the Figure 3 example, SEMU matches the A call with the invocation of B, which reveals the malware's redirection of the system call A to the service B.

To extract the effect of kernel data manipulations in malware behavior, the trace analyzer compares the traces of malware operations both in presence and absence of kernel data protection. For this purpose, whenever a malware sample starts execution, SEMU takes a snapshot of the VM at the original entry point (OEP) of the program. Then it uses this snapshot to run the sample twice. For the first run, the analysis plug-in protects kernel data from manipulations of untrusted codes. In the second round, the plug-in allows the write operations of untrusted code within the kernel data addresses. Then, it compares the two execution logs and reports the differences.

5. IMPLEMENTATION IN QEMU

At a high-level, SEMU uses a plug-in architecture. SEMU's functionality is packaged in components that plug into a VM such as QEMU. This approach decouples SEMU's analysis from the underlying virtual machine, which provides two main advantages. First, SEMU plug-ins can be loaded and unloaded dynamically at runtime to suite the analyst's needs. Second, all the analysis code that is specific to the guest OS or specific to a certain OS version is encapsulated within plug-ins. This architecture makes it relatively easy to support additional versions of the guest OS or a different guest OS such as Linux.

The two main SEMU plug-ins are the VMI plug-in and the analysis plug-in. Execution reaches these plug-ins via callback functions. The VM calls these callback functions before processing certain events such as guest OS system calls, switching from user-mode to kernel-mode, context switches, and kernel heap accesses. In QEMU, memory access operations can be monitored by analyzing the semantics of x86 instructions. For example, we monitor `mov` instructions as they can change the value of a memory region. The x86 language has a vast number of read and write operations. However we can express our analysis very compactly in terms of QEMU's built-in write operation abstractions. QEMU then maps our analysis to all concrete x86 write operations.

Virtual addresses are easy to infer from physical addresses. SEMU therefore stores all addresses as physical addresses, which makes it easy to detect cases in which malware exploits the fact that two different virtual addresses may map to the same physical address. SEMU currently utilizes QEMU's built-in functions for converting virtual addresses into physical addresses.

To monitor read and write operations we customize softmmu codes in QEMU. QEMU uses softmmu in order to convert the physical addresses of the guest system to virtual addresses of the host system. This conversion is needed for each read and write. By hooking into QEMU's softmmu functions SEMU extracts the guest OS address being read or written by the current instruction.

To store the kernel data in the shadow memory, the current SEMU implementation uses the kernel data structure layout definitions of ReactOS[3]. ReactOS is an open-source re-implementation of Windows. But the SEMU code uses these ReactOS layouts only for ease of implementation. That is, we could easily generate these layouts from the PDB files and will do so for future SEMU versions.

6. MIXED-MODE MALWARE SAMPLES

This section describes several samples of mixed-mode malware. These samples serve to evaluate existing and future malware analysis tools.

6.1 Misleading User-Only VMI

This sample implements the Figure 3 motivating example attack that evades analysis by Ether. The sample misleads Ether-style user-only VMI by modifying the semantics of the system calls invoked by the malware, which leads tools such as Ether to log system calls that are different from the system calls actually executed by the malware.

The sample has a user-mode component and a kernel-mode component. The user-mode component Mal.exe is based on the SDBOT malware. We customized SDBOT source code to install our kernel-mode component (Figure 3, step 1) and to make system calls (Figure 3, step 3) after the kernel-mode component has modified the kernel.

Our kernel-mode component (also called rootkit) is a kernel-mode driver packaged as a resource file that changes the semantics of kernel system services. Our rootkit changes the semantics of the `DeleteFile` and `TerminateProcess` system calls.However a different implementation of our rootkit could easily change other system calls.

When the current user-mode thread such as Mal.exe requests a service, the OS follows the `ServiceTable` pointer of the current thread's user-mode `KTHREAD` object to find the address of the requested service (Figure 3, step 4). To change system call semantics, our rootkit manipulates the `ServiceTable` pointer in the `KTHREAD` object of the user-mode Mal.exe process.

For console applications, `ServiceTable` points to the system calls exposed by OS image `ntoskrnl.exe`, via the system call table `SSDT`. That is, the OS initializes the `ServiceTable` pointer once via the internal (non-exported) `KeInitializeThread` function to point to the `SSDT` table.

Our kernel-mode component is somewhat similar to earlier rootkits that operate in isolation, without a cooperating user-mode component. These earlier rootkits set the `ServiceTable` pointer of various threads to the address of a fake `SSDT` table to hide the presence of malware processes [19].

[3] `http://www.reactos.org`

Beyond hiding processes, we manipulate system call semantics to redirect subsequent system calls of Mal.exe in a way that evades current malware analyses.

6.2 MDL System Call Semantic Modification

This sample differs from Figure 3 in that it does not manipulate kernel objects directly. Instead this sample uses OS memory management functions to access and modify the system call table.

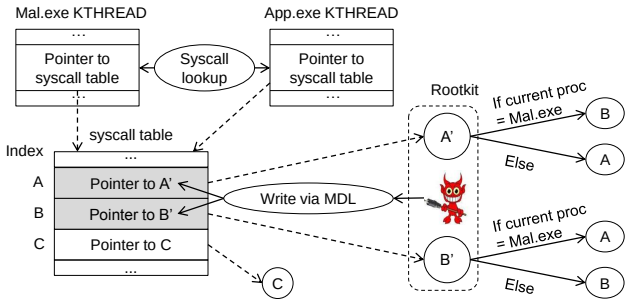

Figure 7: MDL system call semantic modification.

Figure 7 illustrates the kernel-mode component of of this sample. By using standard functions for memory operations, the rootkit creates a Memory Descriptor List (MDL)[4]. A MDL enables the rootkit to map the addresses of the `SSDT` table and overwrite its system call pointers.

In the example of Figure 7 the rootkit has overwritten the original pointers for system services A and B with the addresses of the A' and B' functions. A' and B' are provided by the rootkit. These rootkit functions check the currently running process and swap system calls A and B only if the current process is the malware process Mal.exe.

Similar to Figure 3 this malware can mislead system call tracing. Similar to the sample of Section 6.1 we swap the system calls for `DeleteFile` with `CreateFile` and `TerminateProcess` with `CreateProcess`. The rootkit adjusts input parameters (and return values) in functions A' and B'.

6.3 User-Level Acts on DKOM Attack

This mixed-mode malware sample consists of a user-mode component and a kernel-mode component. The kernel-mode component is based on the kernel-mode component of the FU rootkit, which uses Direct Kernel Object Manipulation (DKOM).

The FU rootkit also has a user-mode component but we replace this component as it only acts as a UI that sends commands to the kernel-mode FU component. Similar to FU, our user-mode component first installs the kernel-mode component, i.e., a kernel-mode driver. Our user-mode component then waits for the kernel-mode component to use DKOM to hide both malware components. The user-mode component then checks if its own process has been hidden successfully and then adapts its subsequent behavior accordingly.

Our kernel-mode component performs DKOM by attempting to hide both malware components by unlinking the corresponding `EPROCESS` and `DRIVER_OBJECT` kernel-level objects in the list of running processes and drivers. Then the user-mode malware component enumerates the current running processes in the victim systems to check if the object hiding attempt succeeded. If the user-mode process is still in the list of running processes the user-mode component injects its payload as a shellcode into the SVCHOST program and terminates. Otherwise the user-mode component continues the execution of its malicious payload.

[4]`http://msdn.microsoft.com/en-us/library/windows/hardware/ff565421(v=vs.85).aspx`

6.4 User-Level Acts on DKSM Attack

This sample differs from the one in Section 6.3 by replacing DKOM with DKSM [1]. This sample uses Direct Kernel Structure Manipulation (DKSM) to swap the image name and process id of the malware process with one of the running processes.

In the Windows operating system, the process id `PID` and the image name `ImagFileName` are member fields of the `EPROCESS` process object. To swap the `PID` and `ImageFileName` of the malware process with a running process, the kernel-mode rootkit accesses the list of process objects by calling the `PsGetCurrentProcess` function. The kernel-mode component can traverse the process list using the `flink` field. Since a kernel-mode driver such as our kernel-mode component can write all kernel memory it is then easy to swap the process id and process name of the malware process with another process.

Similar to the Section 6.3 DKOM sample, if process manipulation does not succeed, malware injects its payload into a victim process in user-mode. Otherwise, the user-mode malware continues execution as a standalone executable.

6.5 Stuxnet's Kernel Exploit

We used the Stuxnet-based example exploit (CVE-2010-2743) of Figure 4 and added a shell-code that performs a privilege escalation attack. The shell-code escalates the privilege level of the malware process, by swapping the `token` fields of the SYSTEM and malware process.

Specifically, the shell-code traverses the list of `EPROCESS` objects of the current running processes, searches for the SYSTEM process, and stores the SYSTEM `EPROCESS` `token` field. It then swaps this token field with the token field of the malware process.

Such a modification makes the malware process execute with administrator privileges. The malware can thus freely invoke a range of Windows APIs that are not allowed for non-privileged users. SEMU effectively detects the execution of the shell-code as an untrusted code running in kernel-mode and logs the modification of the `token` field.

6.6 User-Level Malware Acts on User-Mode Unhooking of Mapped Kernel SSDT

This malware is similar to the Stuxnet sample of Section 6.5 in that it also does not have a kernel-level component. Instead this malware sample has two user-mode components. The first user-mode component performs the tasks of a kernel-mode component, by writing directly into the physical memory pages of the OS kernel. Such memory mapping techniques are commonly used by malware [11]. Although all malware components reside in user-mode, to analyze this malware sample, a malware analysis tool has to keep track of both kernel-space and user-space memory.

For this sample we assume that the system call table (`SSDT`) has been hooked by a malware analysis tool. Our

malware sample thus writes into kernel memory to perform DKOM and unhook the `SSDT`.

Our first user-mode malware component writes to kernel memory by calling the Windows Native API cite[27], which is implemented in *ntdll.dll*. The Native API gives access to the physical pages of the `SSDT` table in the kernel, i.e., via the `NtOpenSection` and `NtMapViewOfSection` functions. Our second user-mode malware component then operates based on the success of the attempted `SSDT` DKOM manipulation.

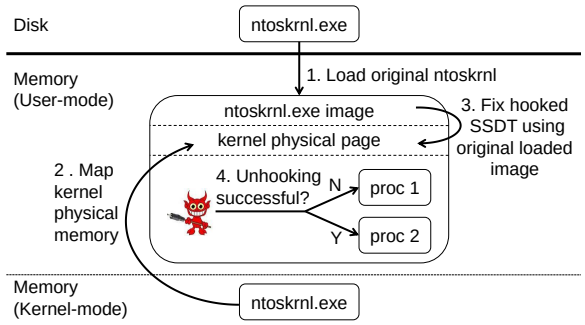

Figure 8: Unhooking system call table by a user-mode malware.

Figure 8 summarizes the attack procedure. As a first step, (1) the malware loads the file of the OS (ntoskrnl.exe) from disk into its user-level memory. This enables the malware to obtain the system service indices from the original SSDT table, which has not yet been hooked by other programs such as a malware analysis or anti-virus software. In step (2), using Native API functions, the malware maps kernel memory to its own address space. In step (3) the malware compares the current SSDT with the original unhooked `SSDT` and fixes the current SSDT in the kernel memory based on the original unhooked SSDT loaded from disk. Based on the success of unhooking the system call table, the malware executes either "proc 1" or "proc 2".

If a malware analysis tool such as a sandbox for user-mode malware analysis depends on `SSDT` hooking for malware analysis then the malware exposes two different behaviors, depending on the success of the attempted kernel manipulation. First, if the malware analysis tool protects the `SSDT` table from being overwritten by malware it can only analyze "proc 1". Second, if the user-mode malware unhooks the `SSDT` successfully then the analysis tool is ineffective for monitoring the rest of the malware execution.

When tracing such a sample with SEMU, we run the sample twice. In the first run we allow the malware to access and overwrite the kernel. In the second run, we prevent the kernel from being overwritten, either directly by malware or indirectly by OS functions invoked by malware. This enables SEMU to analyze both malware behaviors (proc1 and proc2).

7. EVALUATION

To evaluate the SEMU approach of analyzing mixed-mode malware we ask the following two research questions.

- RQ1: Can SEMU analyze mixed-mode malware that cannot be fully analyzed by current state-of-the-art approaches?
- RQ2: Is the SEMU execution time competitive with current state-of-the-art approaches?

To answer these two research questions we first implemented SEMU on top of QEMU as described in Section 5. We then compared our SEMU implementation with the two tools that are both closely related to SEMU and are fully available for experimentation, i.e., tools that provide access to their source code. These two tools were TEMU version 1.0 and the latest available Ether release (version 0.1).

7.1 Analyzing Mixed-Mode Malware (RQ1)

For RQ1, we implemented variations of our motivating examples of Section 3 in Section 6 and analyzed them on TEMU, Ether, and SEMU. We conducted these malware analysis experiments on a Debian Wheezy host system running on a 2.9 G.Hz Intel Core i7-3520M processor machine. The guest OS was Windows XP SP3 with 1 GB RAM 32 bit.

The six malware samples are written in C/C++. Table 2 lists the size of each malware sample in lines of code (LOC). The slowdown numbers in the last column are the overhead SEMU imposes for monitoring the system and writing the system log during malware sample execution. We compare SEMU's overhead with the overhead of competing malware analysis approaches in the following Section 7.2.

The samples perform attacks including DKOM, DKSM [1], and hooking, by manipulating OS objects or data structures such as `SSDT`, `KTHREAD`, `EPROCESS`, and `DRIVER_OBJECT`. In this comparison SEMU was the only tool that could log all the events that are necessary for analyzing these attacks.

7.2 Malware Analysis Execution Time (RQ2)

For RQ2, we compared the total execution times of TEMU and SEMU in Table 3 and of Ether and SEMU in Table 4. To summarize, SEMU was faster than TEMU but slower than Ether. The TEMU/SEMU difference can be explained by the newer QEMU version used by SEMU. The Ether/SEMU difference is due to Ether using hardware extensions. However SEMU also works if these extensions are not available.

7.2.1 SEMU vs. TEMU's Inside-the-Guest VMI

This section compares the performance of SEMU with the closely related TEMU. SEMU and TEMU are built on the same QEMU VM architecture. TEMU uses QEMU v0.9 whereas SEMU uses the newer QEMU v0.14. While the QEMU versions differ slightly, we do not expect a big performance difference from these different QEMU versions.

The main conceptual difference between SEMU and TEMU is the placement of VMI components (partially in-guest in TEMU vs. outside-the-guest in SEMU). The main goal of the evaluation in this section is thus to determine if this change of VMI architecture has a large negative impact on the malware analysis overhead. A malware analyst may be concerned that the switch from in-guest VMI in TEMU to outside-the-guest VMI in SEMU incurs a prohibitive performance penalty.

To compare the performance of in-guest with outside-the-guest VMI, we picked a typical, coarse-grained analysis task (symbol extraction), and applied it on five standard programs. Table 3 lists both the programs and the analysis times of TEMU and SEMU. The performance numbers show that SEMU did not incur an additional overhead over TEMU.

#	Description	Affected Object	Via OS functions	Kernel LOC	User LOC	Slow-down
6.1	Modify system calls	KTHREAD	no	370	1,684	35.3
6.2	Modify system calls (MDL)	SSDT	yes	417	1,684	38.7
6.3	DKOM object hiding	EPROCESS, DRIVER_OBJECT	no	96	451	28.2
6.4	DKSM renaming	EPROCESS	no	111	451	20.6
6.5	Privilege escalation	EPROCESS	no	0	149	25.2
6.6	User-mode unhook	SSDT	yes	0	710	29.1

Table 2: Results of analyzing six mixed-mode malware samples. Via OS functions denotes if the malware manipulates kernel entities directly or by calling OS functions; # = section describing the malware.

Subject	w/o VMI [s]		Coarse [s]		% O/H	
	T	S	T	S	T	S
PsGetsid	1.68	0.56	3.44	1.09	105	95
Pslist -t	3.19	1.03	4.69	1.31	47	27
Psinfo -s	5.76	2.88	9.79	4.78	70	66
Coreinfo	1.70	0.65	3.75	1.07	121	63
ListDLLs	3.20	2.58	5.01	3.75	57	45

Table 3: Performance comparison of TEMU's (T) inside-the-guest VMI vs. SEMU's (S) outside-the-guest VMI using a typical, coarse-grained analysis task (symbol extraction); O/H = Overhead; ListDLLs = ListDLLs -d ntdll.dll.

To make this comparison, we re-implemented the TEMU symbol extraction feature in SEMU, but placed it outside-the-guest with the rest of SEMU. TEMU extracts the names of processes, modules, and exported symbols from a running Windows system. In other words, it keeps track of which processes have which modules loaded at which address, and it enumerates the exported symbols from each module.

To perform the comparison, we execute a Windows batch file in the guest OS that automatically executes and terminates our benchmark applications. The batch file records the application start and termination time stamps.

The guest system in our experiment is Windows XP SP 3 with 512MB allocated RAM. The first column of Table 3 lists benchmark applications from the Sysinternals utilities[5]. The second and third columns are average run-time of the applications in TEMU and SEMU when the guest system runs in its normal state—without VMI. The next two columns are the average application run-times when VMI is active and extracts symbols. The last two columns show the overhead of in-guest VMI in TEMU against our outside-the-guest VMI in SEMU. SEMU exhibited both an overall lower runtime and a lower relative VMI overhead.

7.2.2 SEMU vs. Ether's Single-Domain Analysis

This section compares SEMU's performance with the closely related Ether. Both tools place their analysis components (such as VMI) outside the guest OS.

SEMU differs from Ether in two key aspects. First, while SEMU is implemented on a software virtual machine, Ether leverages hardware extensions. We expect this difference to lead to higher performance in Ether. Second, Ether focuses on a single analysis domain, whereas SEMU covers both kernel-mode and user-mode. We expect this difference to further favor Ether over SEMU in terms of performance.

We conducted this experiment to determine if SEMU's performance remains within the same order of magnitude as the hardware-accelerated Ether. While hardware acceleration is often useful, not every hardware platform offers such acceleration. So it is important to have a an alternative such as SEMU that does not have the hardware constraints of Ether but still provides reasonable performance.

[5] http://technet.microsoft.com/en-us/sysinternals

Subject	w/o VMI [s]		fine VMI [s]		Slowdown	
	Ether	S	Ether	S	Ether	S
Efsinfo	0.63	2.42	20.54	21.39	32	8
Timezone	0.05	0.79	4.41	13.03	87	16
Whoami	0.03	0.72	4.49	19.83	149	27
UPX	0.32	9.00	45.58	322.60	141	35
RAR a	0.15	3.07	45.16	302.93	300	98

Table 4: Fine-grained VMI: Instruction tracing in Ether and SEMU (S). Timezone is Timezone /g.

To compare the performance of SEMU to the hardware-accelerated Ether, we picked a typical fine-grained VMI task, i.e., logging each instruction, and applied it on the five standard programs listed in Table 4. The programs are Windows XP tools and the command-line version of the popular packing and archiving tools UPX and RAR. This experiment was conducted on a Debian Lenny domain-0 OS running on a 2.33 GHz Xeon machine with 32 GB RAM with a 1 GB RAM 32 bit Windows XP SP2 guest OS.

Table 4 also lists the analysis times of SEMU and Ether. SEMU maintains a reasonable performance when compared to the hardware-accelerated Ether.

We expect SEMU's performance to improve in future versions, as we have not yet optimized SEMU for speed. For example, SEMU does not yet leverage QEMU accelerators such as KQEMU [3] or KVM [15].

8. CONCLUSIONS

This paper provided proof-of-concept implementations of malware samples that cannot be fully analyzed by current tools such as TEMU and Ether. To analyze such malware, we presented the SEMU whole-system outside-the-guest analysis tool. The paper compared both analysis capabilities and overhead of SEMU with TEMU on Ether.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 1017305 and 1117369.

9. REFERENCES

[1] S. Bahram et al. DKSM: Subverting virtual machine introspection for fun and profit. In *SRDS*, pages 82–91. IEEE, 2010.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Krügel, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*. The Internet Society, 2010.

[3] D. Bartholomew. QEMU: A multihost, multitarget emulator. *Linux Journal*, (145), 2006.

[4] U. Bayer, C. Krügel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *EICAR*. EICAR, 2006.

[5] U. Bayer, A. Moser, C. Krügel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[6] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS*, pages 555–565. ACM, 2009.

[7] D. Chisnall. *The definitive guide to the Xen hypervisor*. Pearson Education, 2007.

[8] Consumer Reports. Online exposure. *Consumer Reports Magazine*, June 2011.

[9] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *CCS*, pages 51–62. ACM, 2008.

[10] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy*, pages 297–312. IEEE, 2011.

[11] E. Florio. When malware meets rootkits. In *Virus Bulletin*. Virus Bulletin, 2005.

[12] Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy*, pages 586–600. IEEE, 2012.

[13] Y. Fu and Z. Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM TISSEC*, 16(2):7:1–7:29, 2013.

[14] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*. The Internet Society, 2003.

[15] I. Habib. Virtualization with KVM. *Linux Journal*, (166), 2008.

[16] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. C. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *LEET*. USENIX, 2008.

[17] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *RAID*, pages 198–218. Springer, 2007.

[18] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *CCS*, pages 128–138. ACM, 2007.

[19] A. Kapoor and R. Mathur. Predicting the future of stealth attacks. In *Virus Bulletin Conference*, 2011.

[20] C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *CCS*, pages 285–296. ACM, 2011.

[21] A. Lanzi, M. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *NDSS*. The Internet Society, 2009.

[22] A. Moser, C. Krügel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy*, pages 231–245. IEEE, 2007.

[23] M. Neugschwandtner, C. Platzer, P. Comparetti, and U. Bayer. dAnubis - dynamic device driver analysis based on virtual machine introspection. In *DIMVA*, pages 41–60. Springer, 2010.

[24] N. A. Quynh and K. Suzaki. Virt-ICE: Next-generation debugger for malware analysis. Black Hat Briefings USA, July 2010.

[25] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *RAID*, pages 1–20. Springer, 2008.

[26] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *EuroSys*, pages 47–60. ACM, 2009.

[27] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, fourth edition, 2005.

[28] S. B. Schreiber. *Undocumented Windows 2000 secrets: A programmer's cookbook*. Addison-Wesley, 2001.

[29] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, pages 335–350. ACM, 2007.

[30] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *CCS*, pages 477–487. ACM, 2009.

[31] D. X. Song et al. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, pages 1–25. Springer, 2008.

[32] A. Srivastava and J. Giffin. Efficient protection of kernel data structures via object partitioning. In *ACSAC*, pages 429–438. ACM, 2012.

[33] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *CCS*, pages 545–554. ACM, 2009.

[34] C. Willems, T. Holz, and F. C. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[35] C. Xuan, J. Copeland, and R. Beyah. Toward revealing kernel malware behavior in virtual execution environments. In *RAID*, pages 304–325. Springer, 2009.

[36] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *VEE*, pages 227–237. ACM, 2012.

[37] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *NDSS*. The Internet Society, 2008.

[38] H. Yin, P. Poosankam, S. Hanna, and D. X. Song. Hookscout: Proactive binary-centric hook detection. In *DIMVA*, pages 1–20. Springer, 2010.

[39] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, pages 116–127. ACM, 2007.