Re-inforced Stealth Breakpoints

Amit Vasudevan
CyLab
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, USA
Email: amitvasudevan@acm.org

***Abstract*—This paper extends VAMPiRE, a stealth breakpoint framework specifically tailored for microscopic malware analysis. Stealth breakpoints are designed to provide unlimited number of code, data and I/O breakpoints that cannot be detected or countered. However, in this paper we present several attacks that can be used to detect and counter VAMPiRE. We then present a solution towards preventing such attacks in the form of a new breakpoint framework named Galanus. Galanus also adds support for legacy I/O breakpoints in kernel-mode, an important feature required to analyze keyloggers, BIOS flashers, CMOS updaters and rootkits. We also evaluate Galanus, comparing it to VAMPiRE in the context of a few real-world malware.**

***Keywords*-Malware Analysis, Stealth Breakpoints, Debugging**

I. Introduction

Microscopic malware analysis is a fine-grained analysis process that provides insight into malware structure and inner functioning. This helps in gleaning important information regarding a malware to facilitate the development of an antidote. With malware writers employing more complex and hard to analyze techniques, there is need to perform microscopic analysis of malicious code to counter them effectively.

Fine-grained analysis requires the aid of various powerful tools, chief among them being a debugger that enables runtime binary analysis at the instruction level. One of the important services provided by a debugger is the ability to stop execution of code being debugged at an arbitrary point during runtime. This is achieved using breakpoints, which can be of two types: Hardware and Software.

Hardware breakpoints are provided by the CPU and support precise breakpoints on code, data and I/O. They are deployed by programming specific CPU registers to specify the breakpoint locations and type. Software breakpoints on the other hand are implemented by changing the code being debugged to trigger certain exceptions upon execution (usually a breakpoint exception). However, as we discuss in Section II, both hardware and software breakpoints are severely limited in the context of malware.

To address this situation, VAMPiRE [1] introduced a new breakpoint mechanism called stealth breakpoints which provides unlimited number of breakpoints on code, data and/or I/O, that cannot be detected or countered. It uses an approach that exploits the virtual memory system to set breakpoints without modifying the target code in any fashion. VAMPiRE makes it possible to debug and analyze real-world malware that would have otherwise been extremely difficult and/or impossible using traditional hardware and software breakpoint techniques.

Though VAMPiRE is powerful, it is still very vulnerable, especially to malware executing in kernel-mode. Further, it does not contain support for legacy I/O breakpoints in kernel-mode, a feature that is required to analyze malware that do direct I/O access in kernel-mode such as keyloggers, BIOS flashers, CMOS updaters and rootkits. As such malware are becoming increasingly prevalent [2], this is a very important and useful feature for analysis purposes.

In this paper we present several attacks that can be used to detect and counter VAMPiRE. We then present a solution towards preventing such attacks in the form of a reinforced stealth breakpoint framework (codenamed Galanus). Galanus also adds support for legacy I/O breakpoints in kernel-mode. We also evaluate Galanus, comparing it to VAMPiRE in the context of a few real-world malware.

II. Background and Related Work

Breakpoints are a mechanism to stop execution of code during runtime in order to analyze program behavior. Current implementations of breakpoints can be broadly categorized into hardware and software.

Hardware breakpoints are supported by the CPU with or without assistance by a special purpose hardware. Many CPUs have built-in support for hardware breakpoint facilities. This involves a subset of the CPU register set and exception mechanisms to provide precise code, data and I/O breakpoints. Hardware breakpoints can also be supported by using a special purpose hardware acting as a debugging aid. As an example, In-Circuit Emulation (ICE) [3] is a specialized circuitry embedded within the CPU and is used with supporting hardware to provide hardware breakpoints.

Current CPUs however, contain support for only 2–4 hardware breakpoints. This imposes a serious restriction on the analysis process in the context of malware, most if not all of which carry complex polymorphic and metamorphic code streams [4]. Further, since hardware breakpoints were

978-1-4244-4497-7/09/$25.00

designed for normal program debugging, many malware employ efficient anti-debugging primitives to prevent hardware breakpoints from being used altogether. For example, the W32/HIV virus uses CPU debug registers and the breakpoint exception for its internal computations while the W32/Ratos employs the breakpoint exception to handle its decryption in kernel-mode.

Software breakpoints on the other hand provide an elegant, cost-effective and scalable solution with the existing hardware. Practical data breakpoints [5] relies on program source-code and uses efficient runtime data structures and ideas from compiler optimization to provide data breakpoints. GDB [6], Windbg [7] and Softice [8] are a few popular debuggers that use CPU supported trap and/or breakpoint instructions to set code breakpoints. In this method, a 1 byte trap instruction is encoded at the desired breakpoint location. The debugger gets control via the trap exception that the CPU generates upon encountering the trap instruction. Fast breakpoints [9] implements software breakpoints using instruction flow change. The idea is to encode a jump instruction to transfer control to the debugger at a given breakpoint location.

These techniques either rely on the availability of sources or entail modification of the executing code, both of which are unsuitable in the context of malware. Most malware are in binary form and possess self-modifying and/or self-checking (SM-SC) capabilities. Further, these techniques do not support I/O breakpoints. Also, there are speculations regarding the correctness of trap-based breakpoint implementations. The combination of trap and single-stepping may result in missed breakpoints in a multithreaded program if not correctly implemented by the debugger [10].

Another category of software breakpoints is found in debuggers based on virtual machines such as TTVM [11]. However, the virtual machines themselves can be detected [12] or exploited [13] by the malware which can then evade analysis.

III. Stealth Breakpoints

VAMPiRE is a new breakpoint mechanism designed to overcome the limitations of current breakpoint implementations in the context of malware. Breakpoints under VAMPiRE are realized through a combination of virtual memory, single-stepping, TSS (for applicable processors) and simple stealth techniques. The basic idea involves breakpoint triggering by manipulation of memory page attributes of the underlying virtual memory system (for code, data or memory-mapped I/O breakpoints) and I/O port access control bits in the TSS[1] (for legacy I/O breakpoints in user-mode).

[1]Task State Segments (TSS) are supported by all x86 class CPUs and are used to store CPU task information. This includes the CPU registers, stacks, the tasks virtual memory mappings.

Support for virtual memory on most processors, is in the form memory pages. The memory pages can have various attributes such as read-only, read-write, present, not-present etc. These attributes along with a page-fault exception (PFE) is used to provide virtual memory support and memory protection. A PFE is generated by the CPU when a reference to a page is inconsistent with the page attributes (e.g. a write is issued to page with an attribute of read-only). VAMPiRE installs its own page-fault handler (PFH). To set a breakpoint (code, data or memory-mapped I/O) at the desired memory location, VAMPiRE sets the attribute of the page corresponding to the memory location to not-present. This results in a PFE when any location is referenced in that page. The PFE is then processed by the PFH of VAMPiRE to determine if a breakpoint has been reached.

VAMPiRE relies on the TSS to provide user-mode legacy I/O breakpoints. The TSS consists of a bitmap structure called the I/O Bitmap which is a bit array with 1 bit for every legacy I/O port in the system. If the bit corresponding to a I/O port is set to a 1, the processor causes a general protection fault (GPF) when I/O instructions referencing that I/O port are executed. VAMPiRE installs its own general protection fault handler (GPFH). VAMPiREs GPFH semantics is very similar to that of its PFH. To set a breakpoint at the desired I/O location (read or write), VAMPIRE sets the bit corresponding to the I/O port, in the I/O Bitmap array to a 1. This results in a GPF when any access is attempted using that particular I/O port. The GPF is then processed by VAMPiREs GPFH to determine if a target breakpoint has been reached.

VAMPiRE employs a single-step handler (SSH) to implement breakpoint persistence. As an example, when a breakpoint is set at a memory location using VAMPiRE, the entire memory page corresponding to the breakpoint location has its attribute set to not-present. Therefore, access to other locations within the same memory page not corresponding to a breakpoint, will also result in PFEs. VAMPiRE employs a SSH to step over instructions causing such access one at a time. The idea is to set the memory page attribute to present to prevent PFEs for the duration of a single instruction and set the attribute back to not-present after the instruction has been single-stepped. This ensures that future accesses to the page re-triggers breakpoints contained within it.

IV. Detecting Stealth Breakpoints

In this section we present various schemes that a malware can employ to detect stealth breakpoints from both user and kernel-mode.

A. Detection using Memory Page Attributes

VAMPiRE uses the virtual memory system to implement memory breakpoints. For this purpose, it marks the memory page attributes containing the breakpoint as not-present. The memory page attributes are stored in page tables which

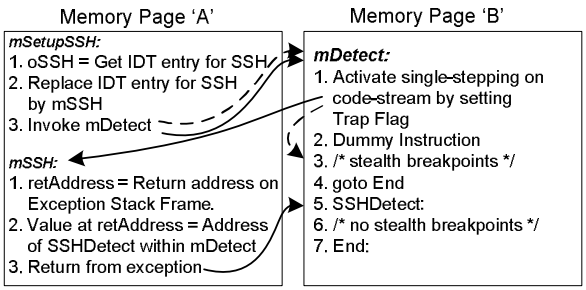

Figure 1. Detection of Stealth Breakpoints using Custom Single-step Handler (dotted lines indicate control flow in presence of stealth breakpoints)

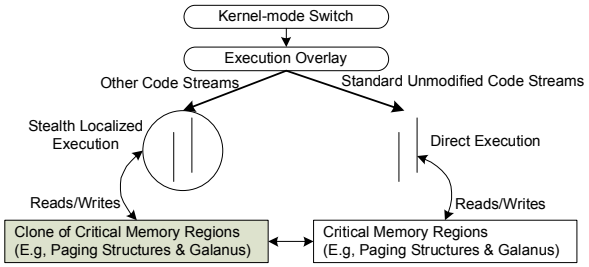

Figure 2. Galanus Kernel Mode Monitor: Protects Galanus from attacks originating in either user or kernel-mode.

are resident in memory and accessible only to kernel-mode code. A malware executing in kernel-mode can allocate all memory pages pertaining to its code and data to be non-pageable. In other words, these memory pages are always present in memory. It can then directly access the page tables and check the attributes of its memory pages. Any not-present memory page attribute indicates the presence of stealth breakpoints.

B. Detection using Custom Page-fault Handler

VAMPiRE installs its own Page-Fault Hander (PFH) during system initialization to implement memory breakpoints. A PFH is invoked by the CPU when the attributes of a memory page is inconsistent with the operation performed. (e.g, write to a memory page that is marked read-only). A malware executing in user or kernel-mode can install its own PFH on top of the existing PFH. In kernel-mode, the Interrupt Descriptor Table (IDT) can be used to register the PFH while OS supplied exception handling mechanisms can be employed (e.g, vectored exception handling under Windows) in user-mode. The malware can then allocate all its memory pages to be non-pageable. Thus, under normal circumstances the malware PFH will never encounter a fault address that belongs to its memory pages. However, when a stealth breakpoint is set on any of the malware memory pages, the malware PFH will see a fault address belonging to the memory page that was marked not-present by VAMPiRE.

C. Detection using Custom Single-step Handler

VAMPiRE uses CPU single-stepping for breakpoint persistence. It installs its own Single-step Fault Handler (SSH) for this purpose. The SSH is installed on demand whenever executing code and breakpoints lie on the same memory page. Let us consider a scenario where a malware code stream is being analyzed in either user or kernel-mode. Consider the pseudo-code of the function mSetupSSH as shown in Figure 1. The function mSetupSSH installs a malware SSH, mSSH on top of the existing SSH by using the IDT. mSetupSSH then invokes the function mDetect which detects if stealth breakpoints are present. Let us assume that both mSetupSSH and mSSH are both on memory page 'A' and mDetect is in memory page 'B'. Line 1 of mDetect sets the CPU trap flag. Thus, after line 2 of mDetect executes, a single-step exeception is generated by the CPU and control is transferred mSSH. Lines 1-3 mSSH modify the exception stack frame to continue execution from line 5 of function mDetect. Thus, during normal execution, control flows from line 2 to line 5 in function mDetect.

Now, let us assume that a stealth breakpoint is set anywhere on memory page B. VAMPiRE will execute function mDetect using single-stepping to ensure breakpoint persistence. When line 2 of function mDetect is stepped over and VAMPiREs SSH gets control, it simply sets the CPU trap flag and returns which causes line 3 of function mDetect to execute instead of line 5. This difference in control flow can be used to detect stealth breakpoints.

D. Detection using Disassembled Fault-Handlers

VAMPiRE installs its own PFH and General Protection Fault Handler (GPFH) to implement memory breakpoints and legacy I/O breakpoints in user-mode. The handlers are not installed directly on the IDT, but are installed by modifying the default OS handlers to transfer control to VAMPiREs handlers. In our experiments with the default PFH and GPFH handlers of various versions of Windows, we noticed very little or no difference in terms of changes made to them (2 changes for Windows 2000 in five service packs, no changes for Windows XP and Windows 2003 Server in two service packs). Given that benign code never modify these default handlers, a bytewise comparison of preacquired OS default handlers to the runtime handlers suffices to detect VAMPiRE.

V. Re-inforced Stealth Breakpoints

As seen from the previous section, all the attacks against vampire is due to the fact that the framework does not really protect against code running in kernel-mode. Thus, the galanus architecture concentrates on protecting kernel-mode execution thereby reducing the attack surface greatly.

A. Kernel Mode Monitor

Galanus adds a new component called the Kernel Mode Monitor (KMM) for this purpose (see Figure 2). The KMM performs two important functions: protects critical kernel-mode structures and OS regions from tampering and facilitates kernel-mode legacy I/O breakpoints. The KMM employs three concepts for its functionality: Stealth Localized Executions (SLE), Execution Overlaying and Cloning.

1) Stealth Localized Executions: Stealth Localized Executions [14] is a mechanism by which a code stream (or a group of code streams) is executed in a manner that allows monitoring/alteration while mimicing its normal execution. In other words, it allows dissecting an executing code stream at the instruction level while making it very difficult for the code stream to detect SLE in a deterministic fashion.

SLE begins at an overlay point and ends at a release point, a range within a code stream where analysis is desired. The basic idea behind the runtime dissection of a code stream involves dynamic binary translation. SLE begins by disassembling instructions at the overlay point to construct instruction blocks within a local code cache. The instruction blocks are then scanned to insert stealth implants which prevent SLE from being detected by the executing code-streams. The instruction blocks are then executed within the least privilege level during which SLE handles various events that a block might cause. E.g, access to specified memory regions, registers etc. Every instruction block terminates with an xfer-stub - a group of non-invasive instructions that transfer control to the next instruction block via SLE.

SLE is completely re-entrant and hence is multithreaded in both user and kernel mode. Further, it incorporates performance enhancement strategies which allow coalescing of instruction blocks as well as skipping standard OS code streams. Most importantly SLE runs on a real system at real time thereby allowing a malware to see the real system as it would normally.

2) Kernel-mode Execution Overlaying: One way for the KMM to execute kernel-mode code with supervison, would be to set an overlay point on every entry into kernel mode and an release point into exit from kernel-mode. However, doing this would entail running the entire OS kernel code within SLE which would result in high run time costs. However, if we assume that we can trust the underlying OS and associated kernel mode components are not malicious to start off with, we can then employ SLE only on kernel-mode code that are not a part of this standard set. Note that this assumption is very realistic as malware analysis is typically carried out on systems which are clean in the first place.

There are only a fixed number of ways in which a non-standard kernel mode code stream can execute within a commodity OS. As an example on all versions of Windows (XP, 2K, 2003) the kernel function KiSwapContext is the only function that can switch to a kernel-mode code stream for execution. The KMM sets a stealth breakpoint at the KiSwapContext function and executes the target non-standard code stream using SLE.

1: $linearAddr \Leftarrow$ linear address of the page-fault
2: $mPfn \Leftarrow$ memory page corresponding to $linearAddr$
3: **if** no active breakpoints on $mPfn$ **then**
4: $pfhPointer \Leftarrow$ current PFH from cloned IDT
5: Apply clock patch
6: **if** $pfhPointer = OSdefault$ and OS regions unmodified **then**
7: Resume execution at $pfhPointer$
8: **else**
9: Start SLE at $pfhPointer$
10: **end if**
11: **end if**
12: Attributes of $mPfn \mid= Present$
13: $eAddr \Leftarrow$ effective address of instruction that caused the page-fault
14: **if** $eAddr = BreakpointAddress$ **then**
15: $bpType \Leftarrow$ get breakpoint type [read, write or execute]
16: $ProcessBreakPoint(linearAddr, eAddr, bpType)$
17: **end if**
18: **if** $CPUInstructionPointer \in mPfn$ **then**
19: $CPUTrapFlag \mid= SingleStep$
20: Setup SSH in original IDT
21: **end if**
22: Apply clock Patch
23: End of Fault

Figure 3. Galanus Page-fault Handler: Handles Code, Data and Memory-mapped Breakpoints

3) Cloning: The KMM maintains clones of memory pages that correspond to critical memory regions such as page tables, descriptor tables, OS code and data and supporting OS kernel-mode components. When Galanus is first initialized, it creates exact copies of such memory pages which are called clones. KMM then sets up SLE in such a way that during block execution all the original pages are marked non-present. Further, SLE is setup to catch read/writes to such memory areas during block execution. Upon access to such regions, the KMM presents to the block the cloned version of the pages. Note that, standard code-streams see the original critical memory regions, while the non-standard code streams always see the cloned copies.

B. Breakpoint Implementation

Breakpoint implementation under Galanus differs from VAMPiRE for memory breakpoints, legacy I/O breakpoints in kernel-mode and for breakpoint persistence. Legacy I/O breakpoints in user-mode are handled the same way as in VAMPiRE, by exploiting the TSS in conjunction with a General Protection Fault Handler (see Section III and VAMPiRE [1]).

1) Memory Breakpoints: Galanus, like VAMPiRE, employs virtual memory and page protections to implement stealth memory breakpoints. However, the difference lies in the way memory breakpoints are triggered within non-

standard kernel-mode code streams and the way in which the existing Page-Fault Handler (PFH) is invoked.

Figure 3 shows the PFH of Galanus. When a memory breakpoint is set using Galanus, the attribute of the memory page where a memory breakpoint is set, is made not-present. This results in a page-fault exception when any location in that page is referenced. When the PFH of Galanus gets control, it first checks to see if there is an active memory breakpoint on the page. If not, the PFH applies a clock patch (to hide its processing latency) and obtains the current PFH address from the clone IDT. If the PFH address defaults to that of the OS and the OS regions are unmodified, the OS PFH is invoked directly. If the OS regions were modified in some form, then the PFH of Galanus sets an overlay point on the current PFH address and begins SLE. This is shown in lines 1–11.

If the PFH of Galanus was invoked due to an active memory breakpoint on a memory page, the effective address of the instruction and the type of memory access causing the fault are obtained. If the effective address matches any breakpoint location and type, the breakpoint is triggered and processed (lines 12–17, Figure 3). The PFH of Galanus then prepares to re-trigger any persistent breakpoints on the memory page by setting the CPU trap flag (see Section III). It then applies a clock patch and returns marking the end of exception (lines 18–23).

2) Kernel-mode Legacy I/O Breakpoints: To support legacy I/O breakpoints in kernel-mode, the KMM configures SLE to implant xfer stubs for all legacy I/O instructions that are found within all block corresponding to the instruction stream that is executed. On the x86 class of CPUs there are only 4 variants of the I/O instructions that need xfer-stubs (the IN, INS, OUT and OUTS instructions). The xfer-stubs transfer control to Galanus which then compares the destination port and triggers the breakpoint.

3) Breakpoint Persistence: Breakpoint Persistence under Galanus, like VAMPiRE, is achieved using the single-step feature of the CPU; a non-breakpoint instruction causing the fault is stepped over while temporarily disabling breakpoints. However, there are two important differences. The first is that the Single-Step Handler (SSH) of Galanus maintains a per thread copy of the CPU trap flag. This is needed in order to invoke the SSH of the thread in case it installs one. Secondly, Galanus checks to see if the virtual CPU trap flag is set in the executing code-stream and if so invokes the thread SSH using SLE.

Figure 4 shows the SSH of Galanus. When the SSH of Galanus gets control due to a CPU single-step exception, the handler first makes sure that any effect of the CPU trap flag (indicator for single-step) is masked out of the instruction that has just been stepped over. This prevents the executing code stream from seeing the original value of the CPU trap flag. The SSH of Galanus then temporarily disables any active breakpoints on the memory page. For user-mode

```
1: remove effect of CPUTrapFlag and update virtualCPUTrapFlag of code-stream
2: for each activeBreakpoint do
3:    if activeBreakpoint is legacy I/O then
4:       IOBitmap[Port] ⇐ 0
5:    else
6:       mPfn ⇐ memory page corresponding to activeBreakpoint
7:       Attributes of mPfn |= Present
8:    end if
9: end for
10: if virtualCPUTrapFlag is SET then
11:    sshPointer ⇐ current SSH from cloned IDT
12:    Apply clock patch
13:    if sshPointer = OSdefault and OS regions unmodified then
14:       Resume execution at sshPointer
15:    else
16:       Start SLE at sshPointer
17:    end if
18: end if
19: Apply clock Patch
20: End of Fault
```

Figure 4. Galanus Single-Step Handler: Handles Breakpoint Persistence

legacy I/O breakpoints, it clears the corresponding bit in the I/O Bitmap. This is shown in lines 1–9.

The SSH of Galanus then checks the status of the virtual CPU trap flag of the executing code stream. If it is set, the SSH of Galanus obtains the current SSH address from the clone IDT. If the SSH address defaults to that of the OS and the OS regions are unmodified, the OS SSH is invoked directly. If the OS regions were modified in some form, then the SSH of Galanus sets an overlay point on the current SSH address and begins SLE. The SSH of Galanus then applies a clock patch (to hide its processing latency) and returns from the exception. This is shown in lines 10–20, Figure 4.

C. Security Analysis

The primary goal of Galanus is to ensure that neither itself or any stealth breakpoints deployed by it are detectable from both user and kernel mode. The KMM employs SLE which in itself cannot be deterministically detected [14]. Further, the KMM employs cloning to ensure that kernel mode code never sees Galanus specific memory regions thereby preventing attacks like the one discussed in Section IV-A. Cloning also ensures that original critical memory regions such as page-tables and descriptor tables are the ones that are always used by the CPU whereas non-standard and modified standard kernel mode code streams read and write to their clones. The clones do not contain values from the original structures that are specific to Galanus (protection flags and exception handlers) and writes to the clones for Galanus specific locations do not persist to the original structures. This prevents attacks of the form discussed in Section IV-B and Section IV-D. Finally, the SSH of Galanus maintains a per thread virtual trap-flag and will chain to the thread

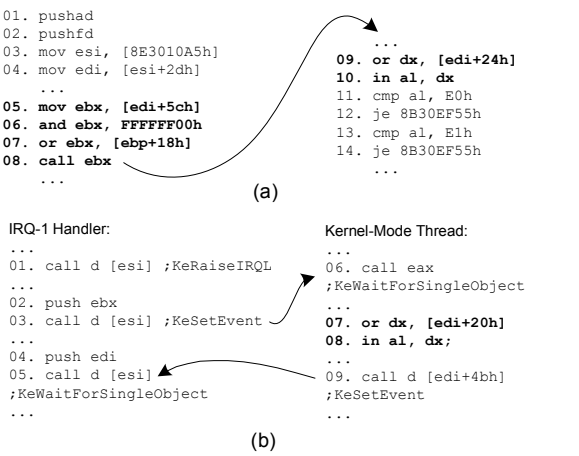

Figure 5. Trojan Feutel-S: (a) employs anti-hardware breakpoint techniques and performs direct keyboard I/O using IRQ-1 handler in kernel-mode, and (b) employs kernel-mode threads to conceal the location where keyboard I/O is done.

specific SSH handler in case one is installed. This mitigates attacks such as the one discussed in Section IV-C.

VI. Experience

In this section we discuss our experience in analyzing two real-world malware, Troj/Feutel-S and W32/Ratos, using Galanus. We obtained the malware samples from Offensive Computing [15]. Our experience shows that kernel-mode legacy I/O breakpoints are a very important tool in the context of kernel-mode malware analysis.

A. Troj/Feutel-S

Troj/Feutel-S is a keylogger running under the Windows OS and is usually installed by other rootkits or remote administration trojans. The malware is registered as a system service (usually named DriverCache) and includes a kernel mode component that installs a keyboard interrupt handler to record keystrokes. Troj/Feutel-S hides its critical files, configuration and process information and has the ability to inject its code into other processes to provide application specific keylogging. In the following paragraphs we describe how we were able to document the keylogging mechanism of Troj/Feutel-S using Galanus.

Before executing the malware, we proceeded by setting a code breakpoint on the OpenSCManager API. The breakpoint was immediately triggered to indicate that the Troj/Feutel-S was loading a kernel-mode driver in the system. We then set code breakpoints on the keyboard class driver dispatch routines to track keyboard I/O Request Packets (IRPs) originating from the malware kernel-mode driver. However, we could not find any such IRPs. We then set legacy I/O breakpoints on ports 0x60–0x64 (keyboard controller ports) with a hunch that the malware was doing keylogging by directly communicating with the keyboard controller. The breakpoints in this case successfully triggered on the code streams as shown in Figure 5a and Figure 5b.

Figure 5a shows part of the Troj/Feutel-S IRQ-1 interrupt handler. This interrupt is generated by the CPU as a result of any keyboard activity. As seen in lines 9–10, the malware uses the IN instruction to record the keystroke directly from the keyboard controller. Also, as seen in lines 5–8, the malware makes use of DR7 debug control register (operand of line 5) to check if any hardware breakpoints are set. If so, the target address of the CALL instruction in line 8 is computed to be a spurious one a leads to an irrecoverable fault in kernel-mode. Such checks are found scattered throughout the malware metamorphic code streams thereby precluding the use of traditional hardware breakpoints.

Troj/Feutel-S also employs keyboard I/O in its kernel-mode threads (Figure 5b). This makes it difficult to find the exact location where the keyboard I/O is done. Let us consider Figure 5b. The kernel-mode thread waits on a kernel event that is set by the IRQ-1 handler (line 6). When IRQ-1 triggers as a result of a keyboard activity, it signals the corrsponding event (lines 2-3), upon which the kernel-mode thread performs the I/O involving the keyboard controller (lines 7–8). Once the I/O is complete, the kernel-mode thread signals the IRQ-1 handler (line 9), which then performs an end of exception returning to the kernel.

B. W32/Wuke

W32/Wuke is a kernel-mode rootkit used by the W32/Wuke@MM worm to hide its files and processes. W32/Wuke incorporates low-level device access to the harddisk and the BIOS and employs several anti-debugging tricks which makes it difficult to analyze using traditional debugging tools. In the following paragraphs we describe how we were able to document the file access mechanism of W32/Wuke using Galanus.

As a first step, we set code breakpoints on all filesystem related APIs in both user and kernel-mode to observe the files that were being created or used by worm. We could immediately observe the W32/Wuke being loaded by the worm. However, after that point the code breakpoints corresponding to the filesystem APIs were never triggered. Our hunch was that, the worm was somehow bypassing the filesystem APIs and directly accessing the harddisk using the W32/Wuke rootkit. We thus proceeded by setting a code breakpoint on the DeviceIoControl API which is used to communicate with a kernel-mode driver from user-mode. We could immediately observe a series of breakpoints corresponding to the DeviceIoControl API issued to the W32/Wuke rootkit.

We then set code breakpoints on the disk driver dispatch routines to track any I/O request packet being sent to the underlying disk. However, we could not find any I/O request packet that were being created bypassing the flow of a

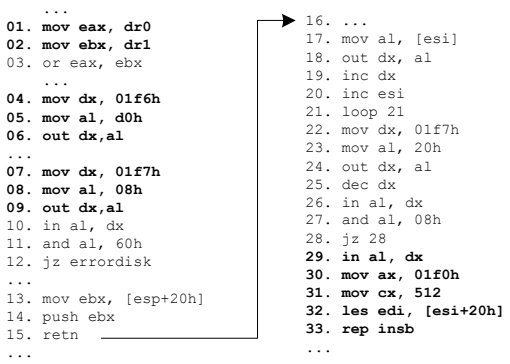

Figure 6. W32/Wuke Rootkit: employs anti-hardware breakpoint techniques and performs direct disk I/O using programmed I/O mode of ATA controllers.

No.	Malware	Mode	Galanus	VAMPiRE	H/w	Type
Memory Breakpoints (code and data on same memory page)						
1	Feutel-S	Kernel	0.314	0.303	NA	NS
2	Wuke	Kernel	0.412	0.399	NA	NS
3	Ratos	User	0.789	0.795	NA	S
4	Mydoom	User	0.397	0.405	0.026	S
Memory Breakpoints (code and data on different memory pages)						
5	Ratos	Kernel	2.122	0.120	NA	S,L
6	Doom	User	0.165	0.155	0.095	S,L
7	Ratos	User	0.864	0.845	NA	S,H
8	Wuke	User	0.102	0.110	0.019	NS,L
9	Doom	User	0.112	0.103	0.011	S,L
Kernel-mode Legacy I/O Breakpoints						
10	Wuke	Kernel	0.367	NA	NA	DevIO
11	Wuke	Kernel	3.215	NA	NA	Init
12	Feutel-S	Kernel	0.004	NA	NA	Thread
13	Feutel-S	Kernel	1.785	NA	NA	IRQ-1

Figure 7. Performance Measurements for Galanus as compared to VAMPiRE and traditional Hardware Breakpoints shows that the breakpoint latency is well suited for interactive analysis. *(NA=Not Applicable, NS=Non-Selfmodifying code, S=Selfmodifying code, L=Light data access, H=Heavy data access)*

normal filesystem function invocation. We then concluded that the rootkit was controlling the harddisk at a much lower level, possibly using direct I/O commands to the harddisk controller. To ascertain this, we set a legacy I/O breakpoint on the command I/O port being used by the primary harddisk controller (0x1F7). The breakpoint triggered in a region of code shown Figure 6(lines 7–9).

Let us consider the code fragment shown in Figure 6 Lines 4-9 program the primary harddisk controller in PIO mode to select the first physical harddisk. A further investigation into the retn instruction in line 15 reveals that it calls a function which corresponds to reading sectors from the harddisk using PIO commands (lines 16–33). Also seen in lines 1-2, is one of the anti-hardware breakpoint strategies employed by the malware which checks for any active hardware breakpoints and if so halts the CPU.

After several such investigations we found that the W32/Wuke upon initialization copies itself and the worm towards the end of the root partition and erases details in the file allocation tables. Thus, the OS does not see any files specific to the worm. When the worm needs to read/write its supporting files, it does so via a DeviceIoControl command to W32/Wuke, which in turn employs PIO mode to access the required information from the harddisk.

VII. Performance Evaluation

The performance of a breakpoint framework such as Galanus, depends heavily on the number and nature of code streams being debugged, and the way breakpoints are set. These factors are not easy to characterize and hence it is difficult to employ a representative debugging session for performance measurements. Therefore, we report the performance of Galanus on debugging sessions involving various real-world malware (W32/HIV, W32/MyDoom, W32/Ratos, Troj/Feutel-S and W32/Wuke).

We use our prototype debugger for performance measurements on an AMD 1.8GHz system with 512MB of physical memory running Windows XP SP2. Figure 7 shows the performance measurements of code, data and I/O breakpoints under Galanus, in the context of various arbitrary analysis sessions involving real-world malware. The code fragments within each session were chosen so that their functionality remained the same with their structure being more or less constant for various deployments of the malware.

The latency shown represents the additional time that has elapsed (in seconds), between executing the code with the breakpoint set and breakpoint triggering. We use wall-clock time as our performance metric as our primary aim is to show that the framework latency is suitable for interactive analysis. Also shown in Figure 7 are the corresponding performance measurements for VAMPiRE and hardware breakpoints for each analysis session. We choose to omit software breakpoints from our performance measurements as they are bypassed by all the malware. Further, certain analysis sessions do not have a representative value for VAMPiRE and hardware breakpoints. These sessions involve analysis of the malware code where neither can be employed.

A. Code and Data Breakpoints

Performance measurements of code and data breakpoints are divided into two broad categories: (a) where the executing code and breakpoint fall on the same memory page, and (b) where the executing code and breakpoint fall on different memory pages.

In user-mode, when code and breakpoints are on different memory pages, the latency is strictly governed by the intensity of data access to the memory page containing the breakpoints. E.g, analysis session 7 has higher latency than analysis ranges 6, 8 and 9. Further, in user-mode, the latency is lower when the executing code and breakpoints are on different memory pages (analyis sessions 6, 8 and 9) than when they are on the same memory page (analysis

sessions 3 and 4). This is due to breakpoint persistence that results in multiple invocations of the SSH when the code and breakpoint are on the same memory page. As seen Galanus and VAMPiRE incur similar overheads in user-mode.

In kernel-mode, when the executing code and breakpoints are on the same memory page, Galanus performs the same as VAMPiRE since the SSH latency in VAMPiRE and the KMM SLE latency in Galanus balance themselves (analysis sessions 1 and 2). In kernel-mode, when the executing code and breakpoints are on different memory pages, the performance of Galanus is more than VAMPiRE due to the KMM SLE latency (analysis session 5). In all cases, the latency of Galanus is well suited for interactive analysis.

B. Kernel-mode Legacy I/O Breakpoints

To measure the performance of kernel-mode legacy I/O breakpoints under Galanus, we set breakpoints on ports 0x60–0x64 and 0x1F0-0x1F7 in our analysis sessions involving the Feutel-S and Wuke malware respectively.

As seen from Figure 7, the latency in analysis sessions 10 and 12 are quite small when compared to the rest. Session 10 involves I/O during DeviceIoControl requests from the Wuke worm to the rootkit while session 12 involves I/O performed within kernel-mode threads of the Feutel-S trojan. Since I/O in both cases is performed soon after the SLE overlay, the KMM only executes a small number of instructions via SLE before triggering the I/O breakpoint. Sessions 11 and 13 on the other hand result in relatively higher latencies. This is due to the polymorphic and metamorphic code-streams in the case of Wuke and Feutel-S respectively. The KMM in this case executes a large number of instructions via SLE before triggering the I/O breakpoint. In all the sessions, the latency of legacy I/O breakpoints in kernel-mode is found to be within limits to suit interactive debugging.

VIII. Conclusions, Limitations and Future Work

This paper presented Galanus, an extension of previous work on stealth breakpoints. We discussed various attacks on the existing stealth breakpoint implementation and showed that it is especially vulnerable to kernel-mode code. We also presented a general solution to the attacks which effectively prevents any user or kernel-mode code from detecting or countering Galanus. Galanus also adds legacy I/O breakpoint capabilities in kernel-mode, a feature whose importance we demonstrate by discussing our experience with a kernel-mode keylogger malware and a rootkit. Galanus thus reinforces stealth breakpoints in the context of malware that are increasingly becoming hardened to analysis.

Galanus is currently ineffective against malware that use hypervisor (e.g, Blue Pill [16]) or System Management Mode (SMM) for execution [17]. However, malware which employ such techniques are rare in practice. We are currently working on addressing this limitation by implementing Galanus as a standalone hypervisor.

References

[1] A. Vasudevan and R. Yerraballi, "Stealth breakpoints," in *Proceedings of 21st Annual Computer Security and Applications Conference*, Dec. 2005.

[2] K. Kasslin, "Kernel malware: The attack from within," *Association of anti Virus Asia Researchers*, 2006.

[3] H. Chen, C. Kao, and I. Huang, "Analysis of hardware and software approaches to embedded in-circuit emulation of microprocessors," in *Proceedings of the 7th Asia Pacific Conference on Computer Systems Architecture*, Jan. 2002.

[4] P. Szor, *The art of computer virus research and defense*. Addison-Wesley and Symantec Press, 2005.

[5] R. Wahbe, S. Lucco, and S. L. Graham, "Practical data breakpoints: Design and implementation," in *Proceedings of the conference on Programming Language Design and Implementation*, Jun. 1993.

[6] M. Loukides and A. Oram, "Getting to know gdb," *Linux Journal*, 1996.

[7] J. Robbins, "Debugging windows based applications using windbg," *Miscrosoft Systems Journal*, 1999.

[8] Compuware Corp., "Debugging blue screens," *Technical Paper*, Sep. 1999.

[9] P. Kessler, "Fast breakpoints: design and implementation," in *Proceedings of the conference on Programming Language design and implementation*, Jun. 1990.

[10] N. Ramsey, "Correctness of trap-based breakpoint implementations," in *Proceedings of 21st Symposium on Principles of Programming Languages*, Feb. 1994.

[11] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the Usenix ATC*, Apr. 2005.

[12] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: VMM detection myths and realities," in *USENIX Hot Topics in Operating Systems*, 2007.

[13] P. Ferrie, "Attacks on virtual machine emulators," *Symantec Advanced Threat Research*, 2006.

[14] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *Proceedings of IEEE Symposium on Security and Privacy*, May 2006.

[15] Offensive Computing. (2009, May) Community malicious code research and analysis. [Online]. Available: http://www.offensivecomputing.net

[16] J. Rutkowska, "Subverting Vista kernel for fun and profit," SyScan and Black Hat Presentations, 2006.

[17] R. Wojtczuk and J. Rutkowska, "Attacking SMM memory via Intel CPU cache poisoning," Invisible Things Lab, 2009.