Virtual Breakpoints for x86/64

Gregory Price[1*]

January 30, 2018

Abstract

Efficient, reliable trapping of execution in a program at the desired location is a hot area of research for security professionals. The progression of debuggers and malware is akin to a game of cat and mouse - each are constantly in a state of trying to thwart one another. At the core of most efficient debuggers today is a combination of virtual machines and traditional binary modification breakpoints (int3). In this paper, we present a design for Virtual Breakpoints, a modification to the x86 MMU which brings breakpoint management into hardware alongside page tables. We demonstrate the fundamental abstraction failures of current trapping methods, and rebuild the mechanism from the ground up. Our design delivers fast, reliable trapping without the pitfalls of binary modification

1 Introduction

In the modern age of malware analysis and "fuzzing for dollars" security researchers are actively searching for a robust, transparent, efficient instruction trapping system. The field is largely moving away from pure-emulation solutions to using virtual machines. Unfortunately, this dive into bare-metal has been plagued with all the normal headaches of debugging, with the added complexity of architecture-level implementation.

Debuggers [1] and other instrumentation tools [3][4][7][11] are becoming increasingly complex as malware deploys equally complex anti-analysis techniques. For example, SPIDER breakpoints [2], rely on caching quirks of Intel CPU's to remain efficient, and dynamic binary instrumentation can be a mess of trampolines and runtime hooking.

Despite the growing complexity, current trapping (i.e. breakpointing) systems must still trade off between efficiency, reliability, and transparency [4]. If the breakpoint methodology introduces significant overhead, analysis becomes tedious. Trapping methods that can be bypassed are becoming less reliable over time. Any system that can cause guest machine corruption is unreliable at best. Unfortunately, each of these problems still exist in modern debuggers today.

Researchers have made headway in solving many of the reliability and efficiency problems [4][2][3][8], but most still rely on some form of binary modification. The difficulty with binary modification for debugging actively hostile programs, is the lack of assurance the modifications are not a) detected, b) bypassed, or c) introducing undefined behavior.

Other approaches, such as single-stepping, debug registers, and emulation have been tried, but all fail to meet the flexibility and efficiency requirements. Single stepping a large, complex program (like an OS) is extremely inefficient [1][2]. Usage of debug registers can be detected by guest machines [2]. Finally, pure-emulation is simply no longer sufficient to run and debug modern operating systems.

Cutting edge x86/64 debugging systems such as SPIDER [2] may accomplish (to an extent) the

**This work was supported by Raytheon CSI

†1 Gregory Price is price.gr at husky.neu.edu

goal of stealth and efficiency, but fail to mitigate the age-old corruption problem with binary modification breakpoints. Binary modification systems carry a presumption of well-behaved programs and lack of user error. For example, if a 5-byte instruction is located at memory address 0x10, and a 1-byte breakpoint is set on memory address 0x12, the resulting behavior could be a range of unintended consequences (Illegal Instructions, Corruption, etc). SPIDER cannot handle the user-error case, and handles the degenerate case by simply deleting the breakpoint (compromising reliability).

In this paper, we propose a memory management unit extension for virtual machine debugging. We claim that the corruption issue is the critical piece of the puzzle that, if solved, will lead to robust, efficient debuggers. The current field of debuggers attempt to fix this fundamentally unsolvable problem by way of building "a better mousetrap", but malware writers eventually learn to escape.

Our novel solution uses a "breakpoint buddy-frame" and adds a "breakpoint bit" to page table entries. When a byte on a page with this bit set is accessed, the breakpoint frame is referenced (in hardware) to determine if that address has a breakpoint set. This breakpoint information is stored on a byte-per-byte basis with the data frame.

For each byte read from a breakpointed page, an 8-bit value is retrieved from the breakpoint buddy-frame and used to determine if an interrupt is generated. This byte implements standard read/write/execute breakpoint settings, and generates a debug break prior to executing the target instruction if the conditions are met.

Unlike binary modification, the breakpoint occurs prior to the execution of a guest instruction, mitigating any possibility of corruption. By doing away with binary modification as the defacto standard of breakpointing, we gain reliability, transparency, and guaranteed correctness of target program execution - all without sacrificing flexibility or efficiency of execution.

2 Background

Debuggers and dynamic analysis tools traditionally implement breakpoints in three different ways: single-stepping, debug registers, and binary modification [1]. Each mechanism is not without their flaws, and much research [3][4][5][6][7][8][11] has been done on the topic of mitigating their issues. Now, in the age of virtual machines, we find there may be a way to fix the true underlying issue.

Despite modern tools trying to mitigate the flaws of classic methods, each subsequent system increases in complexity and reduces efficiency. Furthermore, many still fall prey to anti-debugging techniques such as timing attacks, code integrity checks, and just-in-time compilation.

2.1 Traditional Breakpoint Methods

Each method of traditional breakpointing carries their own unique failures. Each battles with trying to achieve flexibility, efficiency, transparency, and reliability - but none seem to solve for all four.

Single-stepping approaches make execution of a complex process unbearably slow. Typically implemented via use of a flag in the eflags/rflags register, a full context switch between debugger and debuggee is required on each instruction. This can push execution time to be orders of magnitude longer than the original program [2]. Even emulation approaches, which are just a form of interpreter, are simply too slow for general use.

Debug Registers are a finite resource that are not practical nor easily virtualized. On x86 there are only 4 debug registers (DR0-DR4), limiting a host to 4 total watch/break points. Further, because this is a physical register limitation, the debuggee can detect whether these registers are being used [2][10].

Finally, while binary modification meets the requirement of efficient execution, it is easily detected by a debuggee which monitors the integrity of its own code [2][10][5][6]. It is generally accomplished by placing an "int3" (debug break) instruction at the given address. A debuggee that periodically hashes its entire read-only codebase would be able to detect this change, and modify its behavior.

It is still not apparent whether these traditional breakpointing methods can meet the requirements of an "optimal solution". In-fact, we claim these methods break a layer of abstraction that may guarantee they cannot.

2.2 "Stealthy" VM Breakpoints

The original method of breakpointing a virtual machine hinged upon a straight application of binary modification. This carried all the failures of the normal debugger method, namely that it is easily detected. Much research has been done to hide these breakpoints, but only recently has a simple and efficient solution emerged [2].

When Intel and AMD released virtualized MMU support (known as Extended/Nested Page Tables [12]), the idea of transparent binary modification came to fruition with SPIDER [2]. SPIDER introduced a binary modification mechanism that made use of extended page tables to split the "read-write" view of guest data from the "execute" view of data. By doing so, a guest cannot view a breakpoint set via binary modification, because it could only read from a sanitized view.

Unfortunately, this transparent breakpointing system still fails to be sufficiently flexible and reliable. First, it falls victim to the "Critical Byte Problem" which will be described in the next section. Second, because the host must maintain consistency between data and execution views, the guest still has a mechanism (writing to its code pages) with which to affect breakpoints. We will discuss this later.

Moving forward, we will demonstrate that systems relying on binary modification cannot achieve perfect flexibility and reliability. If we hope to accomplish truly transparent and efficient breakpointing, then we must design it from the hardware up.

3 Overview

In this section we discuss the goals of an optimal breakpointing system, and fundamental problems with present-day solutions. We will construct an abstraction of how these breakpoints system are implemented, and identify the evident failures. What we find is that current systems are attempting to solve an impossible problem.

3.1 Goals

We borrow our goals directly from the road map SPIDER[2] set forth, with one modification.

1. Flexibility : *Ability to set a breakpoint or watchpoint at any memory address.*
2. Efficiency : Maintain high performance
3. Transparency : The target program should not be able to detect breakpoints
4. Reliability : The target program should not be able to bypass or tamper with breakpoints.

When we modify flexibility as such, we find that no binary modification solution can accomplish all four requirements. Even the best system, given a degenerate case, falls back on alternate methods which violate at least one. For example, when a debuggee overwrites its own code, SPIDER must evict breakpoints from the modified page to avoid introducing undefined behavior. We have dubbed this issue the "Critical Byte Problem".

3.2 The "Critical Byte Problem"

Binary modification breakpoints depend on the execution of an instruction to trigger a breakpoint interrupt. On x86, this is accomplished by replacing the first byte of an instruction with an int3 instruction (binary 0xCC). This dependence on *execution of guest code* to determine *host behavior* is implicit trust by the host that

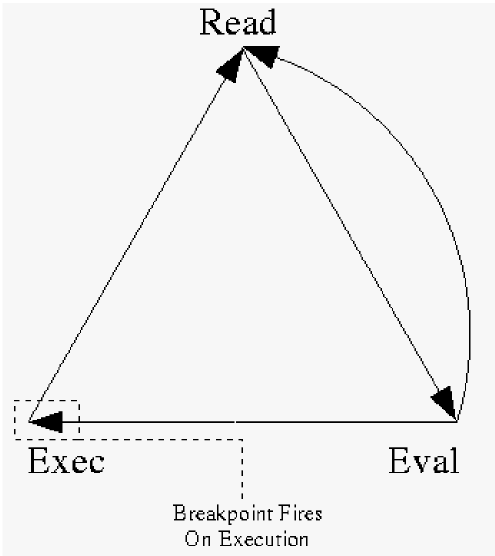

Figure 1: Standard x86 processor Read-Eval-Execute loop. Breakpoints fire on execute.

the binary modification will not introduce undefined behavior.

This causes what we have dubbed the "Critical Byte Problem". If a breakpoint is set on any byte other than the first of an instruction, it will cause undefined behavior. *Figure 2* shows an example of how a jmp instruction can be overwritten with an int3 instruction, or modified to jump to the incorrect place. When the processor reads the instruction with the misaligned breakpoint, the first byte is used to evaluate the remaining arguments. When it reads the 0xE9, it decodes the 0xCC as part of the 4-byte jump address.

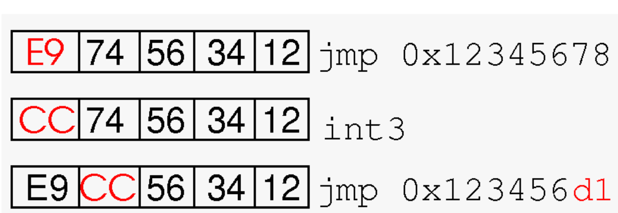

Figure 2: Misaligned breakpoints cause undefined behavior

What the above example tells us is that our "optimal system" should not depend on the execution of guest instructions, nor should it modify guest data. We should avoid trusting the guest to provide debug information, and we should maintain the integrity of guest data to guarantee correctness.

Systems such as SPIDER attempt to build complex solutions to this problem, but only end up trading efficiency for reliability. Further, no binary modification breakpoint system can programmatically re-set a breakpoint over data that has been overwritten by the debuggee, for exactly the reason above.

What we find is that the critical byte problem is a symptom of a larger abstraction failure. The next section will deconstruct these abstraction failures, and show how providing a proper breakpoint virtualization layer might mitigate this problem.

3.3 Breakpoint Abstractions

It's useful to step back and look at how data in a virtualized system is segregated by ownership. Extended Page Tables, as in *figure 3*, create two layers of page tables, where each guest memory reference must go through a second layer of translation to reach the physical memory frame. These page tables are appropriately named the "Guest Page Table" and the "Host Page Table". This simple "ownership" abstraction will reveal the problem.

In *figure 3* we see how a traditional binary modification breakpointing system breaks this abstraction by holding both data and breakpoints in the same data frame. Given this design, the guest can trivially read and interact with host breakpoints. There exists no mechanism with which to prevent this.

When we build a similar abstraction for SPIDER, we find the core feature of the system suffers from the same issue. While the "data view" is pristine and considered wholly owned by the guest, the "execution view" still contains a mix of host and guest data.

Despite not being able to see the breakpoints,

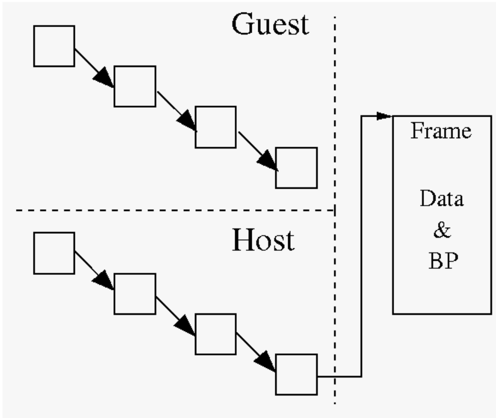

Figure 3: Extended Page Tables still share a single frame for data and breakpoints

the guest can still force an overwrite by blindly overwriting its own code pages. Due to SPIDER's use of separate frames, it must implement a complex system to keep the data consistent. When a breakpoint overwrite occurs it must either fall back to another execution method, use a more complex trapping mechanism, or simply evict the breakpoint all together.

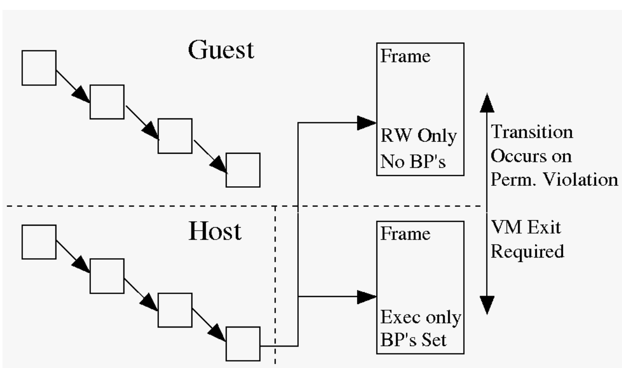

Figure 4: SPIDER also uses a frame containing data and breakpoints

In the following section, we will propose a system which entirely separates guest data from host breakpoints. This simplifies the handling of degenerate cases, mitigates the "Critical Byte Problem", and inherently provides transparency.

4 Design

When modifying the memory management unit of a processor, some care must be taken to extend it in such a way that is reasonably friendly to existing operating systems. Our design is no exception, so we provide an OS modification that makes use of virtual breakpoints as a proof of concept.

4.1 Breakpoint Virtualization Layer

Unlike developing software breakpoint systems, a similar hardware-based system has extremely strict limitations and requirements. We claim any MMU modification that supports a separate view of data and breakpoints must adhere to the following at a minimum:

1. TLB Compatibility
2. Simple Hardware-Friendly Translation Mechanisms
3. Flexible Allocation and Use

To be TLB compatible, any design must be limited to using only the machine physical address and the page size. The use of host virtual addresses is prohibited because the TLB stores only a direct translation from guest virtual to machine physical address[12]. We are afforded the page size from the MMU being set up by the host operating system to walk a predetermined structure.

Next, the mechanism of translation from a data address to a breakpoint address must be simple and fast. Requiring multiple additional memory dereferences or the use of CPU based registers may not be feasible, so a direct translation is preferable.

Finally, the mechanism must be optional. Requiring use on all pages would effectively halve the size of usable memory. While this may be considered a fair trade off for a specialized system, a general purpose system must allow flexible use.

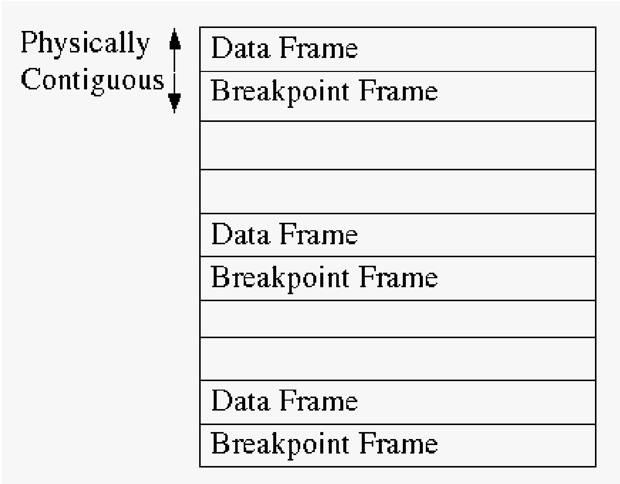

Figure 5: Individually contiguous buddy frames avoid the need for large contiguous areas of memory

When applying the first two requirements strictly, we find a design that uses physically contiguous page frames to be the obvious solution. Implementing buddy frames on a per-frame basis, rather than in bulk, allows us to avoid allocating large contiguous areas memory (*figure 5*).

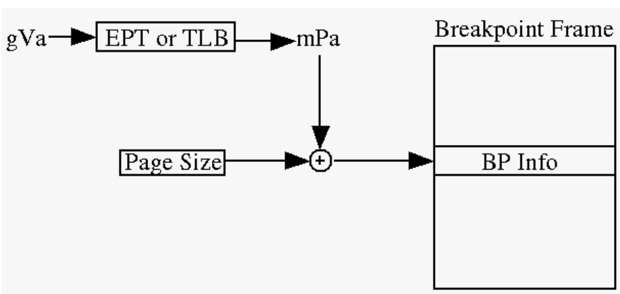

Figure 6: Virtualization layer to translate physical address to buddy frame addresses

The translation from data address to breakpoint address is then a simple addition of the operating system's page-size. This also makes our solution extensible to operating systems configured with pages larger than the standard 4KB (*figure 6*).

The last requirement dictates an agreement between the operating system and the MMU about how page frames will be allocated and managed. This is accomplished through the use of a page table entry which has software-defined bits.

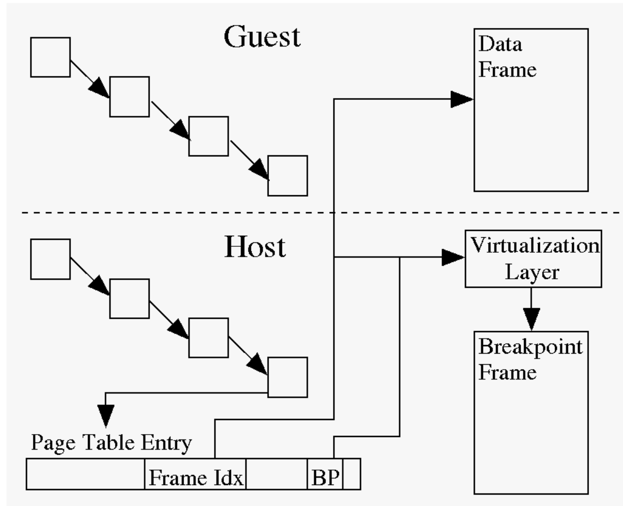

Figure 7: Virtualized Breakpoints. Data is "guest-only", Breakpoints are "host-only".

We propose the addition of a "Breakpoint Bit" to the page table entry *(figure 7)*. Much like the "present bit" determines whether the MMU produces a page-fault, the breakpoint-bit would determine whether the MMU would do a subsequent breakpoint-frame lookup for an accessed address. Breakpoint info would be stored in byte-for-byte parity with the data frame.

4.2 Host OS Modifications

Given the above hardware implementation, the host OS modifications are straight-forward. A host must provide three new mechanisms:

1. Contiguous buddy frame allocation
2. The page table entry "Breakpoint Bit"
3. A breakpoint manipulation API

The first two mechanisms are solved by implementing a new kernel allocator option. This option would allocate two physically contiguous frames, and set the breakpoint bit on the data frame's page table entry.

The final mechanism is a new kernel API which allows for the breakpoint frame to be accessed in a pre-defined manner by the host machine. We implement a per-byte breakpoint API, but a per-page optimization could easily be provided if large breakpoints are needed. This API gives debugger and instrumentation developers a standardized way to set breakpoints, and leaves binary modification systems in place as a backup.

Interestingly, because there is no limitation set on accessibility of guest data by the host, traditional binary modification breakpoints can still be used.

4.3 Analysis of Design

Using the above design, we re-examine the original goals the design sets out to accomplish.

- Flexibility: Breakpoints can be set on any address, and the allocation of buddy-frames is limited to just those pages where breakpoints are set.

  At worst the number of frames can increases linearly with the number of breakpoints. This can double host memory usage. However, we note that SPIDER breakpoints suffer from the same problem [2].

- Efficiency: Guest execution now occurs purely in bare-metal without requiring additional instrumentation for misbehaved programs.

  A guest now executes and performs I/O on the same data frame without incurring a context switch. A context switch on systems such as SPIDER can take thousands of instructions [2], and occurs twice per fault (exit and entry). Instead, our solutions amortizes this cost by distributing a small number of cycles per executed instruction on a target page **(hypothesis)**.

  When the number of breakpoints is small and concentrated in a local region of code (a typical use-case), performance suffers only on a page where a breakpoint is set. Performance is no worse than SPIDER's stealthy breakpoints for self-modifying code **(hypothesis)**, and only marginally less efficient otherwise **(hypothesis)**. As the number of breakpoint frames grow, the number of memory references grows by a factor of at most 2 (one for data, one for breakpoint information).

- Transparency: Complete segregation of guest and host data ensures the guest has no avenue with which to see breakpoint information.

- Reliability: Complete segregation of guest and host data ensures the guest has no avenue with which to affect breakpoint information.

In the worst case scenario, at least one single-byte breakpoint is set on every page of a target. This would double memory usage and cause an additional memory reference per memory-reference to occur. While the memory usage is not trivially mitigated, the memory referencing issue can take advantage of existing caching mechanisms to mitigate the hit. **(hypothesis)**

Furthermore, in the worse-case scenario, SPIDER breakpoints and naive binary modification are still available for use. The host still has access to modify guest data, and SPIDER breakpoints are a fortunate consequence of EPT and independent caches views. This means our extension retains backwards compatibility with existing breakpoint methods.

5 Implementation

5.1 Modify QEMU i386 MMU

- tlb pte breakpoint bit checking
- page table walking bit checking
- buddy frame lookups when breakpoint bit is set
- raise an exception when breakpoint terms are met

5.2 Modified Linux Kernel

- page table entry breakpoint-bit modification
- breakpoint page allocation option/interface for buddy frames
- breakpoint management interface (set breakpoint on given address)
- exception handler (execution/access traps here when exception thrown)
- add memory mapping modifier (take existing non-breakpointed page, create breakpointed version)

5.3 Modified Debugger (GDB?)

- add/remove virtual breakpoint page
- add/remove virtual breakpoint
- add/remove virtual watchpoint
- breakpoint signal handler

6 Evaluation / Experiments

- Overall Efficiency
- Execution Efficiency
- Read Efficiency
- Write Efficiency
- Caching Efficiency
- Anti-Debugging Tests
- Use With Commodity Operating Systems

7 Future Work

7.1 New breakpoint types

- Nesting

  Implementing the virtualization layer such that a guest can set up another layer of virtual breakpointing.

- Booby-traps

  A bit can be set to detect read/write of an address, and automatically break the next time an execute of that address occurs.

- Coverage

  We demonstrate the robustness of this technology via a custom taint-propagation debugger. When external data is injected into memory (via inbound network traffic), the storage location has coverage bits set in its buddy frame. As this data propagates to other areas of memory, the coverage bits populate with them.

- Process Hooking - i.e. "Hook Points"

  A specialized interrupt handler could be registered with the guest which dispatches a "hook point" based on address. The "Hook Point" is set on an address, and the address is entered into a map ccessed by the interrupt handler. Upon executing / accessing the hooked address, the guest exits directly to the interrupt handler for dispatching, then enters back into the guest.

  This could be useful for recording inbound or outbound non-deterministic events for checkpoint/restart and replay systems.

8 Related Work

The idea of segregating "views" of data is not necessarily novel. A number of technologies[2] [5] [6] [13] employ software to modify shadow page tables, extended page tables, and interrupt handlers to mask certain data from guest machines.

Overshadow[13] provides a mechanism with which to hide (encrypt) the contents of a guest-process's memory from the guest kernel. The researchers goals were to provide a mechanism with which a guest process could execute securely, even if the guest operating system was actively hostile. Overshadow accomplished this by extending the original form of page table virtualization released by Intel and AMD - Shadow page tables.

Overshadow implemented a shadow page table containing multiple mappings (encrypted and unencrypted) of a guest's physical memory, and actively tracks the "identity" of the guest process attempting to read a page. When the accessing process does not have the correct identity, an encrypted page is presented (on read) or the machine is terminated (on write). When the accessing process does have the correct identity, an unencrypted page is presented with the permissions originally granted to the page. Unfortunately, its dependence on shadow page tables means it's likely to be insufficiently efficient [14] for modern operating systems and heavy load systems.

VAMPiRE[5] takes advantage of virtual memory page permissions to be executed via alternate methods. In particular, the developers chose to implement a single-step handler which executes any instruction (or accesses any data) falling on a page which contains a breakpoint. This is done to ensure any reads and overwrites of breakpoints are not possible, and so the integrity of guest data is preserved. Unfortunately, relying on single-stepping and what are effectively page-length breakpoints limits the flexibility of the breakpoint system. For example, if a program is contained entirely within a single page, or a breakpointed page contains "hot" code (it's executed regularly), then speed will suffer dramatically.

Finally, Spider[2] comes the closest to implementing a solution that maximizes efficiency, while retaining the flexibility of traditional binary modification breakpoints. Similar to VAMPiRE, it leverages virtual page permissions to determine what "view" of memory should be provided to the processor. On read/write, a "sanitized" view of memory (sans breakpoints) is provided to hardware to prevent detection. On execute, the modified page (containing int3 instructions) is provided directly to the processor.

Spider maintains efficiency by leveraging a quirk in caching mechanisms, which allows both views to be cached (one in the instruction caches, one in the data cache), limiting the number of VM Exits required to expose the correct data on a given access. Unfortunately, as previously discussed, still falls prey to some forms of classic anti-debugging techniques (such as overwriting) due to its reliance on classical binary modification.

Each of these works provide a trend of researchers attempting to split the view of data based on whether we trust the guest. In Overshadow, we wish to trust one part of the guest and not another. In VAMPiRE and Spider, we wish not to trust any attempt to read or write breakpointed areas of memory. This was the primary inspiration for Virtual Breakpoints fully segregated data and breakpoint frames.

9 Conclusion

In this paper we demonstrated the problems with traditional and modern introspection systems to show a clear need for a properly virtualized interface for breakpoints. As we broke down various forms of breakpointing, we identified two fundamental issues with current solutions. Either these solutions were inefficient, or the lacked the ability to breakpoint any address while guaranteeing integrity and correctness of execution.

We proposed the introduction of a memory management unit extension built from the ground up with these goals in mind. Our solution is novel in that it treats guest execution as an untrusted action, and segregates all debug related information into a separate buddy-frame accessible only by the host. Unlike other solutions which make heavy use of hardware virtualization, our system retains integrity of breakpoints and correctness of execution by not requiring the modification of guest data.

We believe that Virtual Breakpointing is the solution virtual machine introspection and interposing systems have been looking for. It presents opportunities for the development of brand new types of breakpoints, and are efficient enough to run commodity operating systems without affecting user experience.

References

[1] Gdb. http://www.gnu.org/software/gdb/.

[2] Deng, Z., Zhang, X., & Xu, D. (2013, December). Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In Proceedings of the 29th Annual Computer Security Applications Conference (pp. 289-298). ACM.

[3] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In EICAR06

[4] D. Bruening. Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, 2004.

[5] Vasudevan, A., & Yerraballi, R. (2005, December). Stealth breakpoints. In Computer security applications conference, 21st Annual (pp. 10-pp). IEEE.

[6] Vasudevan, A. (2009, October). Re-inforced stealth breakpoints. In Risks and Security of Internet and Systems (CRiSIS), 2009 Fourth International Conference on (pp. 59-66). IEEE.

[7] Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008, October). Ether: malware analysis via hardware virtualization extensions. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 51-62). ACM.

[8] Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., & Vasudevan, A. (2012, December). Down to the bare metal: Using processor features for binary analysis. In Proceedings of the 28th Annual Computer Security Applications Conference (pp. 189-198). ACM.

[9] Vogl, S., & Eckert, C. (2012, April). Using hardware performance events for instruction-level monitoring on the x86 architecture. In Proceedings of the 2012 European Workshop on System Security EuroSec (Vol. 12).

[10] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Blackhat USA12.

[11] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In ASPLOS12

[12] Intel, Intel. "Intel(R) 64 and IA-32 architectures software developers manual." Volume 3A: System Programming Guide, Part 1.64 (64).

[13] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., ... & Ports, D. R. (2008, March). Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In ACM SIGARCH Computer Architecture News (Vol. 36, No. 1, pp. 2-13). ACM.

[14] Adams, K., & Agesen, O. (2006). A comparison of software and hardware techniques for x86 virtualization. ACM SIGOPS Operating Systems Review, 40(5), 2-13.

[15] Jeffk.

[16] Albert.