Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks

Stefano Cristalli
Università degli Studi di Milano

Mattia Pagnozzi
Università degli studi di Milano

Mariano Graziano
Cisco Systems Inc.

Andrea Lanzi
Universita' degli Studi di Milano

Davide Balzarotti
Eurecom

Abstract

Spraying is a common payload delivery technique used by attackers to execute arbitrary code in presence of Address Space Layout Randomisation (ASLR). In this paper we present *Graffiti*, an efficient hypervisor-based memory analysis framework for the detection and prevention of spraying attacks. Compared with previous solutions, our system is the first to offer an efficient, complete, extensible, and OS independent protection against all spraying techniques known to date. We developed a prototype open source framework based on our approach, and we thoroughly evaluated it against all known variations of spraying attacks on two operating systems: Linux and Microsoft Windows. Our tool can be applied out of the box to protect any application, and its overhead can be tuned according to the application behavior and to the desired level of protection.

1 Introduction

Memory corruption vulnerabilities are currently one of the biggest threat to software and information security. Education plays a very important role in this area, making programmers aware of common threats and teaching them how to avoid mistakes that may lead to exploitable bugs in their code. However, education alone is not enough, and a good defense in depth approach requires also to put in place multiple layers of mitigation, detection, and exploit prevention mechanisms.

In this field, over the past decade we have witnessed a constant arms race, with the system designers of compilers and operating systems on one side, and the attackers on the other. Over the years, the former have introduced many new security features to increase the complexity of exploiting memory corruption vulnerabilities [31, 6, 41, 9, 40]. This list includes stack canaries [13, 12], data execution prevention (DEP) [2], Address Space Layout Randomization (ASLR) [43, 7, 26, 8], Structured Exception Handling Overwrite Protection (SEHOP) [29], and Control Flow Integrity [3]—just to name some of the most popular solutions. Even though the combination of all these techniques have certainly increased the security of modern operating systems, no matter how high the bar was set, attackers have always found a way to overcome it to take control of a vulnerable system.

ASLR is certainly one of the most common and successful techniques adopted by modern operating systems. In fact, the objective of the majority of memory corruption exploits is to allow the attacker to execute arbitrary code in the context of a vulnerable process. The code can be injected by the attacker herself, or it can be constructed by reusing instructions already present in memory (e.g., in the case of return-to-libc or return oriented programming). Either way, the attacker needs to know where such code is located in memory, in order to divert the control flow of the application to that precise address. And here is where ASLR plays its role: by completely randomizing the layout of the process memory, it makes much harder for the attacker to predict where a certain buffer (or an existing code gadget) will be located at run-time. Unfortunately, attackers found a very simple and effective solution to overcome this protection: fill the memory with tens of thousands of identical copies of the same malicious code, and then jump to a random page[1], hoping to land in one of the pre-loaded areas. This makes this payload delivery technique, called *spraying*, one of the key elements used in most of the recent memory corruption exploits.

Researchers have been looking for ways to mitigate this technique. Unfortunately, the few solutions proposed so far [36, 16, 21] were all tailored to defend 1) a particular application (typically the JavaScript interpreter in Internet Explorer), 2) using a given memory allocator, 3) in a specific operating system, and 4) against

[1] Often a fixed address located on the process heap.

a single form of heap spraying. This made these solutions difficult to port to other environments, and unable to cope with all possible variations of heap spraying attacks. In fact, the original heap spraying attack is now just the tip of the iceberg. The technique has rapidly evolved in different directions, for example by taking advantage of just in time compilers (JIT), by focusing on the allocation of pools in the OS kernel, or by relying on stack pivoting to spray data instead of code. We strongly believe that the increased adoption and sophistication of heap spraying techniques clearly demonstrate the need for a general and comprehensive solution to this problem.

In this paper we present Graffiti, a hypervisor-based solution for the detection and prevention of all known variations of spraying attacks. We decided to implement our solution at the hypervisor level to obtain the first *OS-independent*, *allocator-agnostic* approach to track memory allocations that does not depend on the knowledge of the protected process, or system. By leveraging a novel micro-virtualization technique, Graffiti proposes an efficient and OS-agnostic framework to monitor memory allocations of arbitrary applications. The system is modular, and relies on a set of plug-ins to detect suspicious patterns in memory in realtime. For example, we developed a set of different detection modules based on statistical inference, designed to precisely identify all known spectrum of spraying attacks known to date. Moreover, while all the previous techniques [36, 21] focused on the defense of a particular application or memory allocator against a single form of heap spraying, our system offers the first general and portable solution to the problem.

Graffiti also offers a hot-plugging capability and therefore it can be installed on-the-fly without rebooting the machine and without modifying the native operating system. Our experiments, conducted both on Linux and Microsoft Windows, show that Graffiti has no false negatives and low false positives, with an overhead similar to the one of previous, much more limited, solutions.

In summary, our work makes the following contributions:

- We present the principles, design, and implementation of an effective real-time memory analysis framework. On top of our framework, we developed a set of heuristics to detect existing heap spraying techniques. To the best of our knowledge, we are the first to present a general, efficient, and comprehensive framework that can be applied to all modern operating systems and all existing applications.
- We propose a novel micro-virtualization technique that allows Graffiti to monitor the entire system in terms of both processes and kernel threads, with low overhead.
- We have developed a prototype tool, and performed an experimental evaluation on several existing real-world spraying techniques. Our experiments show that the system is able to detect all the classes of spraying attacks we analyzed with low false positives and acceptable performance.
- We released the source code of the current Graffiti prototype, which is available at the following link: `https://github.com/graffiti-hypervisor/graffiti-hypervisor`

The rest of the paper is organized as follows. Section 2 provides background information on spraying attacks. Section 3 provides preliminary notions about Intel VT-x technology. Section 4, Section 5, Section 6 describe our solution from an architectural point of view. Section 7 reports results on evaluating Graffiti. Section 8 discusses about the security evaluation of our system. Section 9 compares our work with other relevant research and Section 10 discusses future directions and concludes the paper.

2 Spraying Attacks

Heap spraying is a payload delivery technique that was publicly used for the first time in 2001 in the `telnetd` remote root exploit [44] and in the eEye's ISS AD20010618 exploit [15]. The technique became popular in 2004 as a way to circumvent Address Space Layout Randomization (ASLR) in a number of exploits against Internet Explorer [46, 47, 38].

Since 2004, spraying attacks have evolved and became more reliable thanks to improvements proposed by Sotirov [42] and Daniel [14] for a precise heap manipulation. Spraying can now be classified in two main categories, based on the protection mechanisms in place in the target machine: *Code Spraying* and *Data Spraying*. If Data Execution Prevention [2] (DEP) is not enabled, the attacker can perform the exploit by directly spraying the malicious code (e.g., the shellcode) in the victim process memory. On the other hand, when the system uses the DEP protection, the attacker would not be able to execute the injected code. To overcome this problem, two main approaches have been proposed: (a) perform the heap spraying by taking advantage of components that are not subjected to DEP, such as Just in Time Compilers (JITs), or (b) inject plain data that points to Return Oriented Programming (ROP) gadgets. While the internal details between the three aforementioned approaches

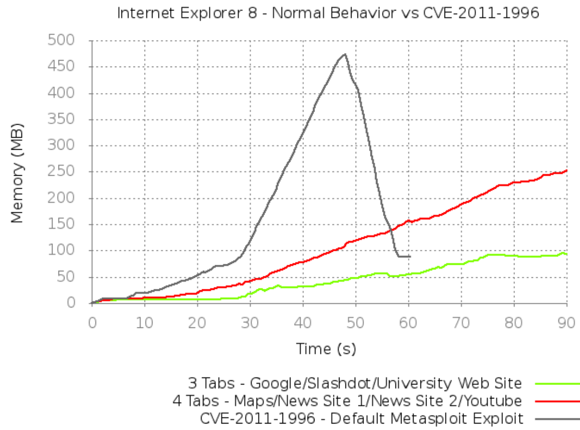

Figure 1: Heap Spraying attack

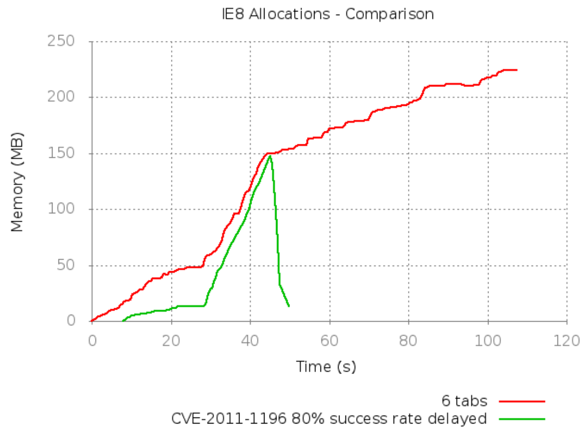

Figure 2: Mimicry attack

may be quite different, what is important for our research is that all these techniques share the same goal, i.e., to control the target dynamic memory allocation in order to obtain a memory layout that allows arbitrary code execution in a reliable way.

It is important to note that spraying is still a valuable technique also in x86_64-based operating systems. In particular, this is the case for user-after-free vulnerabilities – but spraying can also be used in conjunction with vulnerabilities in the ASLR implementation [10], in particular types of vulnerabilities [20], or because of the wide adoption of 32bit processes in 64bit operating systems (as recently shown by Skylined [39]).

2.1 Memory Footprint

The first characteristic of a heap spraying attack that comes to mind is the large amount of memory that is suddenly allocated by a process. Therefore we could erroneously believe that this unusual behavior alone (i.e., many pages allocated in a very short amount of time) could be sufficient to implement a solution to detect spraying attacks. For example, a simple approach could measure the speed of memory allocation and the average amount of memory usually allocated by the application under analysis. The first parameter would react to a quick memory increase, a common aspect of most of the existing attacks. The second parameter, once properly tuned for the application to protect, would act as a threshold of memory allocation, beyond which the behavior becomes suspicious and an alert is raised. It seems reasonable to believe that, by checking these two parameters, a detector could successfully prevent spraying attacks.

One of the main motivation of our work is to prove that the use of these two parameters is not sufficient for designing an effective spraying detection system. To prove our point, we designed a set of experiments to show that an attacker can tune the memory allocation behavior of an exploit to mimic the one of a normal application.

In our tests we used as a case study a classic heap spray attack against Internet Explorer 8 (described in CVE-2011-1996) but it is possible to replicate similar results with any applications where the memory allocation depends on input data. The first test we performed aimed at measuring the memory allocation curve while the user was visiting a small set of web sites. Figure 1 shows that the parallel execution of four common web applications (using parallel browser's tabs) boosts the memory allocation of Internet Explorer to around 200MB. The same graph also shows the allocation curve of the CVE-2011-1996 exploit launched by Metasploit. In this case, the malicious behavior is easy to detect since it produces a huge allocation of memory in a short period of time – that then drops drastically after the success of the exploit. The drop is due to the fact that the shellcode spawns a process and releases the system resources of the previous execution thread along with its own memory. Other spraying attacks exhibit similar curves, a weakness that could be used to identify an ongoing malicious activity.

In the second experiment we wanted to answer two separate questions: i) how the total amount of memory allocated by the exploit affects the reliability of a spraying attack; and ii) whether it is possible for an attacker to slow down the attack in order to mimic the slope of the allocation curve observed on benign web pages. To

this end, we first modified our exploit to decrease the amount of sprayed memory. As we expected, reducing the number of allocated pages also reduces the probability of landing on one of them, thus making the exploit less reliable. We measured this phenomenon by running each exploit configuration ten consecutive times, counting in each case the number of successful attacks. The results of our tests show that the memory used by the original exploit can be largely reduced maintaining an acceptable success rate. For instance, the attack was still successful in 80% of the cases with a total memory consumption of only 131 MB – that is considerably less than what IE8 used in our benign scenario.

We then modified again the original exploit, this time introducing a delay between each memory allocation to mimic the behavior of a benign application. This change had no impact on the success rate of the attack. Figure 2 shows the allocation curve of our modified exploit, compared with a base line obtained by running Internet Explorer with six open tabs. From this experiment, it is clear that neither the speed nor the amount of memory can be used as the only criteria to detect a potential heap spraying exploit. By setting the threshold too low, the system would generate too many false alarms, and by raising the threshold too high the system would be vulnerable to evasions.

This conclusion motivates our further investigation to design a better memory monitoring and spraying attacks detection technique.

3 Preliminary Notions on Intel VT-x

Before we discuss our solution, we need to briefly introduce some virtualization concepts that we will use in the rest of the paper. Intel VT-x is a technology available in various Intel CPUs to support virtualization [23, 30].

VT-x defines two particular *transitions*: `vmexit`, to move from the guest to the hypervisor, and `vmentry`, to move in the opposite direction. As a result, the hypervisor is executed only when particular events in the guest trigger an exit transition. The set of events causing these transitions is extremely fine grained and can be configured by the hypervisor itself. Such events include exceptions, interrupts, I/O operations, and the execution of privileged instructions (e.g., accesses to control registers). Exits can also be *explicitly* requested by in-guest software, using the `vmcall` instruction. Because of its similarity to system calls, this approach is commonly called *hypercall*. Whenever an exit occurs, the hardware saves the state of the CPU in a data structure called Virtual Machine Control Structure (VMCS). The same structure also holds the set of exit-triggering events that are currently enabled, as well as other control information of the hypervisor.

Another technology we need to introduce is the *Extended Page Tables* (EPT). This technology has been introduced to support memory virtualization, which is the main source of overhead when running a virtualized system. If enabled, the standard *virtual-to-physical* address translation is modified as follows. When a software in the guest references a virtual address, the address is translated into a physical address by the Memory Management Unit (MMU). However, the result of this operation is not a real physical address, but a *guest physical address* (*gpa*). The hardware then walks the EPT paging structures to translate the *gpa* into a *host physical address*, that corresponds to the actual physical address in the system memory. The EPT technology also defines two new exit transitions: *EPT Misconfiguration* and *EPT Violation*, respectively caused by wrong settings in EPT paging entries and by a guest attempting to access memory areas it is not allowed to. By altering the EPT entries, the hypervisor has full control of how the guest accesses physical memory. For example, it can remove write permissions from an entry, so that any write-access by the guest triggers a violation.

Threat Model

Our threat model considers an attacker that is able to exploit (either locally or remotely) an application running on the machine and to perform a spraying payload delivery. The use of a hypervisor-based technology is motivated by the goal of providing an OS-independent detection system and a more secure reference monitor.

Since we leverage late-launching to deploy our solution on operating systems running on physical machines, without requiring a reboot, we assume that the machine to be protected is clean when Graffiti is loaded. Thus, we consider the protection of already infected systems to be out of the scope of this paper.

4 Architecture Overview

In order to considerably improve over the state of the art, we set five main requirements for our detection system. First, it should be completely independent from the memory allocator used by the protected applications (R1). Second, it has to operate system-wide, i.e., it should be able to detect any memory allocation and deallocation that occurs in the system (R2) and it must be able to recognize any memory page that gets executed in the operating system (R3). Fourth, in order to operate correctly, our system should not require any OS-

dependent information (R4). Finally, the overhead introduced by the system should be reasonable, in line with other system-wide protection mechanisms. In particular, we consider a “reasonable” overhead, anything comparable to the one introduced by other virtualization systems such as XEN or VMware (R5).

To satisfy these five requirements, our system was designed to be easily extensible and configurable, and to tune its behavior (and therefore its overhead) to match the current level of risk of the monitored system. This is achieved by using two separate modes of operation.

Our monitoring platform is based on a custom hypervisor that normally runs in what we call *monitor mode*. In this mode, the hypervisor intercepts every new memory page that is allocated in the system, along with the CR3 register associated to the process that is requesting the memory. Whenever the total amount of memory requested by a single process exceeds a certain threshold (computed experimentally as described in Section 2.1) the system switches to *security mode* and starts performing additional checks to detect the presence of a possible attack for that particular application (while remaining in monitor mode for the other applications). In Section 2.1 we proved that a fixed threshold is not able to properly capture all the possible variations of spraying attacks. For this reason, in our system we use a threshold not for detection, but only to improve the performance of the system by disabling expensive checks when the total memory used by the process is too low for an attack to be successful. It is important to stress that in our solution, lowering the threshold for a given application does not introduce any false positives from the detection point of view, but only increases the overhead for that particular application alone (and not for the rest of the running system or for any other application).

When a process exceeds this minimum allocation threshold, the hypervisor performs two main tasks. First, using the EPT, it removes the execution permission from all the allocated pages, so that any attempt to execute code will be intercepted by the system. Second, it invokes the static analyzer component to check for the presence of a potential spraying attack. The actual detection is delegated to a configurable number of analysis plugins.

Figure 3 provides an overview of the system architecture and shows the interactions among the different components. The figure is divided in three parts, with user space on top, kernel space in the middle, and our custom hypervisor at the bottom. When an application (in this case a web browser) requests new memory, the kernel searches for a free page and it allocates it. At this point, when the OS tries to update the page table, the operation is intercepted by our hypervisor. If our system is

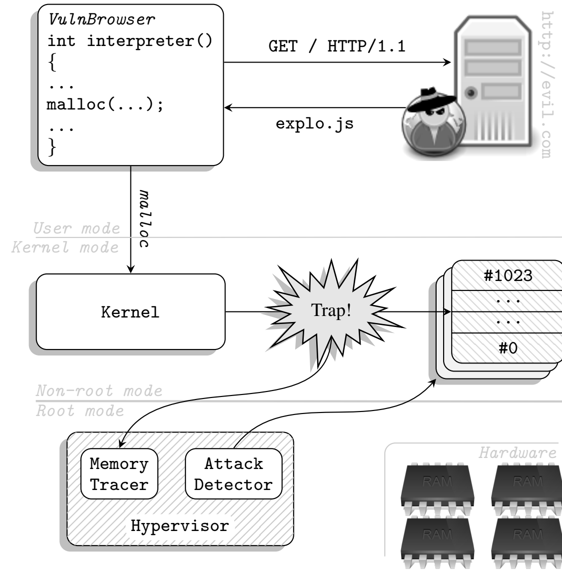

Figure 3: Architecture of the Memory Allocation Tracer.

running in monitor mode, the hypervisor only tracks the new memory allocation and gives back control to the operating system. If instead the application has already requested enough memory to trigger the security mode, our attack detection routines are executed to inspect the memory and flag any heap spraying attempt.

System Deployment

The main component that enables the protections enforced by Graffiti needs to keep an accurate track of all the allocation and deallocation operations that occur in the system. The main motivation of using a hypervisor is that, from a low level perspective, memory allocation is strictly dependent only on the hardware architecture, and not on the operating system itself. Thus, by working *below* the operating system, Graffiti avoids all the intricacies introduced by the various allocation engines, and therefore it does not require to modify or instrument the protected system (e.g., to place hooks inside OS components). Graffiti leverages late-launching to load its protection mechanism while the target is running. This hotplug capability is achieved without rebooting the system, so it is transparent to the native OS. Finally, it is important to note that Graffiti is a very flexible system and can be configured according to the target needs. For instance, it can be deployed to monitor only a single sensitive process (e.g., a browser, or a PDF viewer), a set of thereof, or even the entire running system.

Our current prototype is implemented as an extension of HyperDbg, an open-source hardware-assisted hypervisor framework [17]. In Sections 5 and 6 we present

the design and implementation of the two main components of the system: the *Memory Tracer* and the *Attack Detection Routines*.

5 Memory Tracer

To implement our heap spraying protection technique, we must first keep track of all the allocation and deallocation operations that occur inside the system. Ideally, the most obvious solution to track memory allocations would be to modify the allocator itself, by extending the operating system with a new tracking feature. By doing so, however, our system would need to be customized for a particular operating system, and we would need to constantly update our tracker according to any OS upgrade.

To avoid this problem, we decided to implement our tracking approach at the hypervisor level (requirements R1 and R4), i.e., below the operating system. Since our approach is based on virtualization, from now on we will refer to the protected system alternatively with the term *guest* or *target*.

5.1 Tracer Design

Our system is designed to intercept every modification that is made by the guest OS to paging structures, and to recognize when the change corresponds to the creation or to the elimination of a page. To better illustrate our tracing technique, we will often refer to the paging structures that are used in the Intel architectures [23].

Whenever a process requires a new page, the kernel walks the paging structures of the requesting process looking for a usable Page Table Entry (PTE) in one of the Page Tables of the process (i.e., the second level structures). If none is found, it either allocates a new Page Table, by altering an entry on the first level paging structure (also known as the Page Directory), or it swaps some of the pages of the process to disk to create some empty slots. Once it has found or created a usable PTE, the kernel modifies it to map the allocated physical page to a virtual address, sets the lower 12 bits of the PTE to match the attributes of the page (e.g., `read/write`, `user/supervisor`), and returns the virtual address to the requesting process.

Our defense mechanism needs to keep a fine-grained view of every allocation to protect the system against spraying attacks. In particular, according to the address translation and new page allocation we need to intercept six different events: (1) Creation (2) Modification and (3) removal of a page. (4) Creation (5) Modification (6) Removal of a page table.

Whenever one of these six events is triggered by the kernel, our hypervisor intercepts the operation and acts accordingly. The first triple of events is traced to keep track of which pages a process allocates. The second group, on the other hand, must be traced to ensure that our system maintains a complete view of the allocated pages and does not miss any event in the first category.

5.2 Page Table Monitoring

Since Graffiti operates at the hypervisor level, it leverages the EPTs to write-protect all the page structures of a process. By doing so, it can intercept all modification attempts, as part of any of the six cases enumerated above. At first, the hypervisor detects when a new process is created by intercepting write operations to the `CR3` register. As soon as a it is spawned by the kernel, a process will have just a limited number of paging structures, possibly inherited by its parent process (e.g., on Linux this depends on the flags of the `clone()` syscall that is used to spawn the process). To protect all its paging structures, Graffiti needs to traverse the page directory (pointed by the value of the `CR3` register) and write-protect all the page tables pointed by each PDE. Page tables are scanned as well, to keep track of the physical pages allocated to the process by the kernel. After this setup phase is completed, each attempt to modify one of the pages would cause a trap in our hypervisor system.

Implementing the approach we just described while maintaining an acceptable overhead is a challenging task. At first, we use the EPTs to write-protect every paging structure of a target process. By doing so, whenever the OS kernel attempts to modify such structures because the process requires it (1), an EPT violation transfers the execution to the memory tracer component of our hypervisor framework (2). The violation is handled by removing the write protection and keeping a copy of the value of the entry (PTE or PDE) being modified (3), and re-executing the faulting instruction by performing an entry with the monitor trap flag (MTF [23]) raised (4). After the instruction has been executed, the hypervisor obtains again the control thanks to the exit caused by MTF (5), compares the new value stored in the entry with the old one and uses this information to infer which of the six kernel operations described previously has occurred (6). Eventually, the protection is restored (7) and the control is given back to the guest kernel (8).

To make the tracing mechanism clearer, consider the following scenario: the hypervisor intercepts a write attempt to the 2nd PTE of the 1st page table. This PTE originally contains the value 0. After single-stepping through the write instruction, we collect the new value of the PTE: `old:0x00000000 new:0xcaffe007`.

This means that the guest kernel is mapping a physical page (at address `0xcaffe000`) with a `rw` permission and making it accessible to both user and kernel space. In fact, the three lowest bits are set, making the entry present, writable, and accessible to user mode processes. For our framework, this operation corresponds to a *create page* event. To intercept when a process is created, we catch `CR3` write operations in the guest. When the `CR3` value that is going to be written corresponds to the one of a process we want to protect, we apply the protection to its paging structure, as explained above. It is important to stress that our approach is completely OS independent, as the only knowledge we rely on is the meaning of the bits stored in the paging structures, and those solely depend on the CPU architecture.

5.3 Graffiti Micro-Virtualization

The system described so far does not satisfy the requirement R2. In fact our solution should be able to monitor the entire system, and not only a few processes at a time. Unfortunately, by extending the previous approach to the whole guest operating system (all user-space processes and kernel threads), we observed a thrashing [4] phenomenon that introduced a large overhead in the memory allocation. This phenomenon creates a large number of context switches between OS and hypervisor, thus increasing the system overhead.

This phenomenon happens when a modification of a memory page of the running process creates as a side effect a modification of a memory page of another non-running process. This is a consequence of the fact that some memory pages are shared among processes, and some kernel tasks perform operations on memory pages of different processes. We refer to this problem as the *interference problem*.

The impact of this interference can be measured by running two simple tests. In the first, we computed the overhead introduced by our system while protecting a single process (Internet Explorer 10) and in the second we protected other two processes (Acrobat and Firefox) on top of Internet Explorer. The overhead on Internet Explorer alone went from 22% in the first test to 63% in the second, just as a side effect of monitoring two additional applications. Unfortunately, the interference of protecting more processes and the kernel itself would quickly slow down the entire system to a point in which it would not be usable anymore.

Ideally, we would like to design our system to avoid the interference problem, so that the overhead would not depend on the number of monitored processes. To achieve this goal, we propose a novel *micro-virtualization* technique, where each process runs inside its own virtual memory sandbox and our tracking system enables the memory protection of just the process which is currently running. More in details, our micro-virtualization technique bases its approach on the fact that the VMCS contains a pointer to the EPT (EPTP) currently used by the hypervisor (see Section 3). Since we use the EPT to protect the processes (as explained in Section 5), our idea is to create a different EPT for each of the processes we protect, and change the EPTP in the VMCS at every context switch. From a low level perspective, this corresponds to intercepting every `CR3` write operation (also easily trappable through VT [17]) and modifying the VMCS so that the EPTP points to the EPT of the process that has been scheduled for execution. Protected processes will have their own EPTs, while un-protected ones will just use a common EPT. To this end, every time a new process is created, the system creates a new EPT and associates it to the new process. It is important to note that the creation of this new EPT is not very costly, since the page table at the process creation is tiny and we only need to identify and protect some of them. By using such a mechanism, the hypervisor automatically disables the memory tracking of the other unprotected processes and enables the trapping only for the pages that are related to the currently protected processes, thus avoiding the thrashing side effect. Since this solution requires only to change the EPT pointer when a context switch occurs, it does not increase the overhead of the system.

In order to validate our micro-virtualization mechanism we performed two main experiments by using three applications: IE10, Acrobat Reader and Firefox. During our first experiment we only protect one application (IE10) and we compute the execution time and the overhead obtained by surfing several web pages in three main cases: (1) without hypervisor (2) with out hypervisor but without micro-virtualization and (3) with hypervisor and micro-virtualization enabled. From this first experiment the micro-virtualization does not introduce any additional overhead to the system when is used to protect a single process (23% in both cases with and without micro-virtualization). The only overhead introduced by the micro-virtualization occurs during the first loading of the new process. In this case the hypervisor needs to build up the EPT table for the new process by walking the process page tables. The overhead introduced during the loading time is 8%.

In the second experiment we test the scalability of our system with the new micro-virtualization mechanism enabled. This time we protect all three applications and we compute, like in the previous experiment, the execution time and the overhead obtained by surfing several web pages in the same three main cases: (1) with-

out monitoring the application (no hypervisor enabled) (2) with hypervisor but without micro-virtualization and (3) with hypervisor and micro-virtualization enabled. The overhead was 63% without micro-virtualization and 23% with micro-virtualization, confirming that the micro-virtualization is able to remove the overhead introduced by the *interference* problem.

As a result of our novel micro-virtualization architecture, our system is able to monitor an arbitrary number of different applications, without any increase in the system overhead. More specifically the overhead only applies to a particular protected application and it does not propagate to the rest of the system. For instance, if the user wants to protect only the browser and the PDF viewer against heap spraying attacks, any other application would not suffer any side effect or slowdown from our tracking system.

6 Detection Components

Whenever the total memory dynamically allocated by a process raises over a certain configurable threshold, the tracer switches to security mode and triggers a configurable number of static analysis routines to verify if a spraying attack is ongoing in the system.

Our current prototype includes three different components, presented in details in the next sections. These serve only as possible examples of the heuristics that can be easily plugged into our platform, and they could therefore be improved or extended with other techniques.

Malicious Code Detector

The aim of this component is to detect the simplest form of heap spraying. In this case, we assume the heap is randomized but executable, and therefore the attacker can spray the memory of the vulnerable target with multiple copies of a shellcode. Thus, the goal of this detector is to identify the presence of shellcodes inside the memory allocated by a process.

Our technique works as follows. First, the detector scans a fraction n of the most recently allocated memory pages and tries to disassemble them starting at twenty randomly selected offsets. For simplicity, any sequence of assembly instructions that terminates with a control transfer instruction that invokes a library call or system call is marked as a potential shellcode. To avoid cases where an attacker tries to obfuscate its attack by using an indirect control transfer instruction (iCTI), we consider each iCTI as a potential shellcode terminator.

The scan process is repeated for each allocated page and the detector finally reports the distribution of the number of potential shellcode detected in each page. If the average number is higher than a given value, it raises an alarm. This approach derives from the observation that in the normal operation of a benign program only a small portion of the *analyzed* memory pages would contain a relevant fraction of valid instructions sequences. In an exploitation scenario, instead, most of the analyzed pages would contain close to 20 potential shellcode sequences. It is important to note that when the system starts disassembling from one page it continues till it reaches a code pointer, that may as well be located in a different page. If multiple pages are involved in such analysis they are all considered and marked as a shellcode container.

Self-unpacking Shellcode Detector

In this second scenario, we assume the same environment described before (ASLR enabled, DEP disabled), but we now consider the case in which an attacker packs her shellcode to make the detection more difficult. For example, all Metasploit payloads in spraying-assisted exploits are packed by default, e.g., by using the *shikata-ga-nai* encoder. Packed shellcodes are typically made up of a number of seemingly meaningless bytes, prepended with a small unpacking routine. The routine and the packed code are usually adjacent (i.e., they are located in the same memory page), as splitting them would lead to a waste of space and consequent loss of effectiveness when mounting the spraying attack.

Our second detection plugin is designed to detect packed shellcodes as soon as they start unpacking, and is tightly binded to the memory tracer. The component enforces what we call a *dynamic W⊕X protection*. As soon as the memory tracer detects a new page allocation, it modifies the EPT entry corresponding to the newly allocated page so that a violation will be triggered when a write access to that page is attempted (`R-X`). The detector intercepts these attempts and modifies the EPT entry of the accessed page so that write accesses are enabled, but not execution accesses (`RW-`). If this new protection triggers a violation, we have a *write-then-execute* situation, which is fairly common in nowadays systems (especially with JIT engines). However, this mechanism allows to observe the more anomalous situation in which code modifies the same memory page in which it resides, that indicates the presence of self modifying code, used by packed shellcodes as described above. This technique is also effective when DEP is enabled on the heap memory, and the attacker uses a JIT-spraying attack. In fact, if the JIT-sprayed payload is packed, it will need to unpack itself and thus will trigger our detection heuristic.

Data Spraying Detector

When DEP is enabled, and JIT spraying is not a viable solution (e.g., there is no JIT engine in the vulnerable process), a possible exploit solution is to use return oriented programming. In this case, the attacker no longer sprays the heap with executable code but instead with multiple copies of a ROP chain. To trigger the code, the attacker then uses a *pivoting sequence* to move the stack pointer into the heap and let execution slide down the ROP chain, as we explained in Section 2.

To detect data spraying attacks, we designed a component that samples the most recently allocated memory pages of a process, and it considers any word inside them as a potential memory address. For each of these candidate addresses, the data spraying detector checks whether this address points to a valid executable page, and, if so, marks it as a *potential code pointer*. In case the total number of code pointers for each page is over a threshold, the system raises an alarm.

Unfortunately, even though this policy may sound reasonable at a first glance, we observed that in practice it suffers from a large amount of both false positives and false negatives. The first problem is related to the fact that modern operating systems use different techniques to load pages, one of which is called *Demand Paging* [4]. In this case the pages are only brought into memory when the running-process demands them. This optimization creates an issue for our detection method because when the system extracts the potential code pointers from the memory pages and checks if they point to a valid code page, the page may not be present in the page table (even if it is properly allocated). We observed this behavior during our experiments, and the result is that certain addresses would be discarded—thus potentially creating *false negatives* by missing a page that is part of a spraying attack.

To avoid this issue, we modified our hypervisor to intercept page faults in the guest system when Graffiti switches to security mode for a given process. When the detector checks an address that points to a memory page that is not mapped, the system does not discard it but keeps it as a potential code pointer into the memory structures of the hypervisor. Afterwards, when the process gets access to the demanded page, the system loads it and our detection system intercepts the page faults and it checks if the potential code pointer points to this memory page. If true, the system marks all the memory pages previously allocated that contain such address as suspicious and then it re-applies again the previous technique on the new set of pages.

The second problem of our original technique is the high number of false positive we observed in the experiments because benign memory pages also contain a significant number of code pointers (e.g., in case of C++ classes or arrays). To reduce the false positives created by those benign memory pages, we improved our detector algorithm by replacing the pointers counter with a more sophisticated pointers frequency analysis. The idea is to compute the frequency of the code pointers that *every* page of the entire set contains, instead of analyzing every page individually. While the absolute number of code pointers may be deceiving, we observed that the distribution of those pointers in case of benign applications is really diverse, while in case of an attack the distribution tends to be quite uniform.

7 Experimental Results

The goal of our experiments is to first measure the overhead of the system in a realistic environment and then to show how effective our heuristics are in distinguishing spraying attacks from a normal allocation behavior.

Our code is composed by three main software components: a core hypervisor framework based on HyperDBG, the micro-virtualization implementation, and the detector plugins. The core hypervisor framework is written in a combination of C (17353 LoC) and assembly (545 LoC). The micro-virtualization and detector components account for 1435 lines of C programming language.

All tests presented in this section were performed on two machines, equipped with an Intel Core i5-2500 @ 3.3 GHz and 8GB of RAM, running respectively Windows 7 Professional 32bit and Debian Wheezy 32bit (kernel 3.2).

Activation Threshold and Overhead

Our system is designed to be adaptive. Consequently, the only part that is always active is the Memory Tracer. Our micro-virtualization solution confines the overhead to a single process and allows our system to monitor an arbitrary number of different applications without any increase in the overhead of the rest of the system.

During normal operation, the tracker overhead is negligible, and it is only noticeable when the monitored application allocates tens of megabytes of memory at a time – typically at start up or when a large document is open. To measure this worst case scenario, we used the `stress` suite to simulate a program that intensively allocates memory on a Windows 7 and on a Linux 3.2 hosts at a rate of 8MB every 2 seconds. The overhead we observed during the allocation phase was of 24% on Windows and 25% on Linux for a single process without considering context switch. Again, it is important to

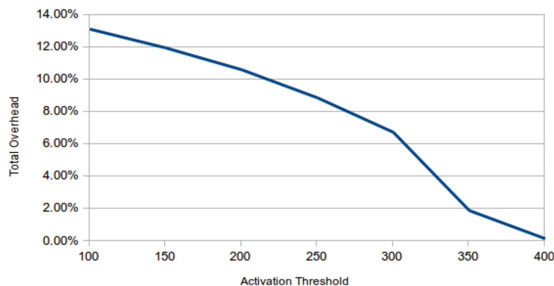

Figure 4: Detection Overhead for Internet Explorer 8

note the experiments performed in these tests produced a very intensive memory allocation activity and it is not representative of the memory behavior of the entire life of a process.

On top of this overhead, each application can observe a different overhead when Graffiti switches to security mode and enables the detection modules to scan the application memory. The frequency at which this happens depends on the value of the activation threshold. The lower the threshold, the hardest it is for an attacker to evade detection – but the higher the *potential* overhead for the application. "Potential" in this context means that the actual overhead also depends on the application: some use so little memory during their normal operation that the security mode would never be triggered - also for very low values of the threshold. Moreover, most of the applications only allocate large amount of memory when the user opens a document, but then the use of memory becomes quite constant - and therefore Graffiti's negative impact tends to be concentrated only on the few initial seconds, and becomes negligible after that.

To measure this trade-off we performed two experiments. In the first, we asked some users to surf the web by using Internet Explorer 8 on Windows 7 with our detection system activated. We choose IE8 since this application usually uses a large amount of memory and it represents one of the main targets of spraying attacks. To mimic a realistic behavior, the users kept a tab open on GMail, and then alternately opened three other tabs performing memory intensive activities: watching videos on `YouTube`, browsing `Facebook`, and checking hundreds of pictures on `9gag`. In the second experiment, we used Acrobat Reader for Linux to open 100 benign PDF files including conference papers, books, Ph.D. dissertations, and very large manuals (i.e., the Intel Manuals).

Following the approach used by Nozzle [36], we selected a sampling rate of 10% (number of pages checked by our detection module over the total number of pages allocated). As a reference, with this value Nozzle introduced an overhead of 20% to Internet Explorer. The overhead obtained with our system is shown in Figure 4 for different values of threshold. Also in the worst case with the activation threshold set to 150MB, the overhead was only 12%. Moreover, the heuristic responsible for most of this overhead is the one that requires to randomly disassemble the content of the memory pages. Since this component is useless on any modern OS when DEP is enabled, the detection overhead of Graffiti becomes barely noticeable.

Moreover, our experiments with Acrobat Reader never reached the activation threshold, even when it was set at the conservative value of 100MB. In this case, the overhead of Graffiti on the normal use of the application was constantly zero – showing that for some popular applications our framework can provide a very complete protection against known and unknown attacks with no additional overhead.

Detection Accuracy

To test the effectiveness of our system, we measured the true and false positives rates for each individual detection technique that is currently included in the Graffiti prototype. To test the detection rate we used several real world exploits that cover all the different spraying techniques and variations mentioned in this paper. It is important to note that the six attacks that we chose for our experiments, summarized in Table 3, are representative for the entire spectrum of the techniques used by the spraying attacks described in Section 2. On top of this qualitative test, we also performed a quantitative test using over 1000 different malicious PDF documents that rely on heap spraying in the exploitation phase.

In the first test we show the effectiveness of our system to detect exploits based on stack pivoting, by using the attack described in CVE-2011-1996. The attack first sprays the stack frames on the heap and then executes a number of ROP gadgets in order to disable the DEP protection. During the spraying phase the attack allocates on average 384MB.

In this case, the static analyzer applied the code pointers frequency analysis on the attack memory pages. The component detected a high number of code pointers with a variance close to 0 in all the allocated memory pages, and thus it raised an alert successfully preventing the attack. To evaluate the false positive of such technique, we instructed our detector to track all the memory pages allocated by Internet Explorer 8 while browsing the first 1000 top Alexa domains [1]. In this case, the frequency of code pointers had a very high variance on all memory pages captured by the system, thus generating zero

Web Domain	Average	Variance
amazon.com	3	259.30
ask.com	7	867.90
baidu.com	8	559.57
blogspot.com	2	158.88
craiglist.org	6	391.15
delta-search.com	8	809.21
facebook.com	23	3521.68
google.co.jp	10	562.99
google.com.br	7	459.57
google.com	10	46.44
instagram.com	14	2763.22
microsoft.com	5	395.72
msn.com	16	2916.28
yahoo.com	14	1183.43

Table 1: Code Pointer Frequency Analysis Results.

Web Domain	Shellcode per page	Average Guess Offset
amazon.com	10/500	1/20
ask.com	2/500	1/20
baidu.com	48/500	4/20
blogspot.com	64/500	4/20
craiglist.org	10/500	2/20
delta-search.com	8/500	3/20
facebook.com	25/500	4/20
google.co.jp	162/500	3/20
google.com.br	69/500	3/20
google.com	74/500	3/20
instagram.com	224/500	5/20
microsoft.com	0/500	—
msn.com	5/500	1/20
yahoo.com	13/500	1/20

Table 2: Shellcode Frequency Analysis Results.

false alarms (Table 1 reports the results for the first 14 domains analyzed).

In the second set of experiments we tested the effectiveness of the Malicious Code detection component. In this case we used the exploit for CVE-2009-2477 affecting the Javascript interpreter of Mozilla Firefox 3.5. This attack exploits a memory corruption vulnerability in the Firefox browser, in which the Javascript interpreter fails to preserve the return value of the `escape()` function and results in the use of an uninitialized memory area. During the exploit, Graffiti reported that the average number of analyzed pages containing a potential shellcode was 100% – thus raising an alarm and stopping the attack. To test the false positive of the same detection technique, we used the same browser to visit the top 1000 Alexa domains. In this case, the average number of potential shellcodes per page was always below 50% and therefore no false alert were raised in the test. In table 2 we reported results about the first 14 domains analyzed, the number of pages that present potential shellcode and the average on shellcode found in the first 500 allocated memory pages. As we can see from the table, our malicious code detector component does not present any false positive.

CVE	Application	Exploit Technique	Detected
2010-0248	Adobe Flash player	ROP + packed sc	Yes
2011-0609	Adobe Reader	JIT + packed sc	Yes
2011-2462	Adobe Reader	ROP + packed sc	Yes
2010-2883	Adobe Reader	Ret2Lib + packed sc	Yes
2011-1996	IExplorer	ROP	Yes
2009-2477	Firefox	Plain Shellcode	Yes

Table 3: Exploitation Detection Results.

In our third experiment, we analyzed the Self-unpacking Shellcode Detector component. In this case we selected different CVEs and we used the metasploit [35] tool to exploit them with packed payloads. In particular, we used two packing methods: the shikata-ga-nai packer and a simple xor algorithm. Our detection system was always able to intercept the first execution of the packed code and consequently detect the attacks without any false negative. Also for this component, we tested the false positive rate by browsing the top 1000 domains from the Alexa dataset. We did not observe any false positive, even though several website included obfuscated Javascript code. A further investigation on obfuscated java-script shows that the de-obfuscation routine is implemented at the compiler level so it does not present any problem or generate any false positive in our system.

To conclude, Table 3 reports all the vulnerabilities we used for our tests, along with the type of payload delivery and the detection results of Graffiti. Even though our detectors had a very high precision in all our tests, an attacker equipped with knowledge about the internals of our detectors could try to mimic the behavior of a benign application to evade detection. A further analysis of such attacks is presented in Section 8.

Aggregated Experiments

So far, we tested each piece of our infrastructure in isolation. In our final experiments, we put all pieces together. In the first test, we used Graffiti to analyze three datasets: a set of 1000 malicious PDF documents, a set containing 1000 benign web pages, and one containing 1000 benign PDFs. The first dataset was collected by a company working on malware analysis, while the other included the top Alexa web pages and random documents collected from various sources. All experiments were conducted with a very conservative activation threshold of 150MB and a sampling rate of 10%. Graffiti successfully detected all malicious documents, with zero false alarms. Moreover, the overhead on loading the web pages was in average of 23% (a value in line with previous OS-specific approaches that were only able to protect the web browser).

In the second experiment, we asked real users to use a Graffiti-protected system during their everyday activities for a total of 8-to-10 hours per day in a 7-days period. Graffiti was installed on two Windows 7 machines, configured to monitor Internet Explorer 8 with an activation threshold of 150 MB. All three spraying attacks detectors were enabled during the experiments (even though the first was not necessary on this setup). Overall, the real users visited a total of 492 distinct web pages and the detectors were activated 55 times, with an average of $\approx$8 times per day. On the same period, Graffiti raised 12 alerts on pages that seemed to be benign. A closer inspection of the FPs showed the data spraying detector (Section 6) to be the only responsible. This component is in charge of detecting data spraying attacks and bases its detection on the number of potential code pointers present in memory.

It is important to stress the fact that the three detection plugins are not the main contribution of our work, and our micro-virtualization framework allows other researchers to easily improve, extend, and replaced them with other techniques. For instance, a possibility to decrease the false positive rate of this component could be to check not only if the code pointer points to a correct executable page, but also whether it points to a dangerous machine instruction sequences (e.g., a gadget). We manually inspected the websites that raised the false alarms and we found that applying such simple method would be able to prevent all the alerts. This check could be activated only when the data spraying detector identifies a possible attack, to prevent a significant increase of the overhead.

8 Security Evaluation

It is possible that an attacker, knowing the internals of our three detectors, could mount a mimicry attack that can successfully evade detection. For instance, an attacker can elude the code pointers frequency analysis by mimicking the variance of benign memory pages. Although this technique can be successful, it has two restrictions. The first is related to the minimum number of gadgets that the attacker needs to connect to perform a useful attack. To be useful, an attack should execute either an API call or a system call. Based on the number of API call parameters, we estimate that a useful number of gadgets for a standard shellcode is around 20 (i.e., to call the `VirtualProtect` function, commonly used in Windows shellcodes to remap a page as executable) even though some previous works show that the length of the gadgets for useful shellcode may vary from 8 to 12 [32]. The second restriction is related to the maximum number of gadgets that an attacker can include to build a shellcode. Theoretically this number could be infinite. In case of spraying attacks, to increase the chance of success, the size of the shellcode should be smaller than the size of the NOP-sled, otherwise the probability to divert the control-flow of the application to the appropriate entry points decreases. In our experiments, for benign applications the range of code pointers in memory varies between 0 and 1024, with the vast majority of pages on the left end of the scale. These values are hard to mimic in a real attack. It may be possible in certain particular cases, but still our component would have considerably raised the bar making the exploitation much more difficult.

Another way an attacker can avoid detection is by evading the shellcode frequency analysis. To this end, the attacker can act on two parameters. She can decrease the number of successful entry points for each page – but this would drastically decrease the success rate of the attack. A second, more subtle, technique would consist in spraying the memory only with NOP instructions, and inject only one copy of the shellcode in a second time, when Graffiti already concluded its analysis. In this case we could extend our component to postpone the analysis when long nop sequences are identified. It is important to note that the attacker cannot wait for a long time in order to inject the shellcode, since any additional memory allocation done by the application would break the continuity of the nop sled.

A current limitations of Graffiti is that it cannot handle the case when an application allocates a big chunk of memory at the beginning of the process and then uses its own allocation functions to perform memory operations. However, since none of the applications that we tested in our experiments exhibited such behavior, we left this case for a future improvement.

To summarize, we believe that evading our three heuristics is not easy but it is certainly possible. However, the contribution of this paper is not in the heuristics per se, but in the underlying monitoring framework. Graffiti offers the first comprehensive, multi-OS solution, and this is an important step forward compared with existing defense solutions and compared with other techniques presented in previous papers.

9 Related Work

Several solutions have been proposed so far to cope with single instances of the spraying problem. In the following we summarize the existing works that address heap, JIT, and data spraying techniques.

Heap Spraying

Researchers have proposed several approaches for detecting heap-spraying attacks [36, 16, 21]. For example, Egele et al. [16] used x86 emulation techniques to defend web browsers against drive-by download attacks that use heap-spraying code injection. More in details, the authors proposed to check for the presence of a shellcode by monitoring all the strings that are allocated by the JavaScript interpreter. Their goal is similar to that of NOZZLE [36], which uses static analysis of the objects on the heap to detect heap-spraying attacks. In particular, NOZZLE scans memory objects looking for a sequence of instructions that includes a NOP sled and ends with a malicious shellcode. However, as the authors point out, the tool presents several drawbacks. For example, attackers can evade detection by avoid using large NOP sleds. Moreover, NOZZLE is also specific for the Java Script Engine Memory Allocator and it cannot be applied to a generic application. Another work to defend against heap spraying attack is BuBBLE [21]. In this case, the authors start from the assumption that an attack needs to spray a large part of the Heap memory with homogeneous data (i.e. NOP sled). BuBBLE breaks such an assumption by inserting special values in a random position inside strings before storing them in memory, and removing them when a string is used by the application. Again this solution is specific for the Javascript language and it cannot be easily ported for the protection of other applications.

Our approach is different since it does not require to know how the memory allocator of a particular interpreter engine works, and consequently it does not require access to source code and it is operating system independent. Moreover, it can protect any system application as well as kernel subsystems without any assumption about internals of the protected component.

JIT Spraying

Bania [5] proposed a detection technique based on the fact that in order to force the JIT compiler to generate code, an attacker should use ActionScript arithmetic operators. However, it is not mandatory for JIT spraying attacks to use arithmetic operations.

Another JIT spraying defense has been proposed by Hu et al. [22]. This solution consists of a kernel patch, `JITsec`, that tests for several conditions when a system call is invoked. In particular, the authors argue that an application can maintain its security properties and execute code from the stack and heap by decoupling sensitive from non-sensitive code and allowing the latter to run from writable memory pages. As a result, such detector only detects attacks that directly issue system calls. Mimicry attack and ROP attacks are therefore not covered by this model.

JITDefender [11] is another work based on hardware-assisted technologies which aims at defeating JIT Spraying attacks. The system protects the Virtual Machine dynamic memory pages created by the JIT-Compiler and allows to execute only the pages requested by the VM. This approach is strictly VM dependent, and it can only detect JIT-spraying attack.

Our solution is orthogonal to the type of attack, and therefore it can successfully detect JIT-spraying attacks without any assumption about the instructions that are used by the attacker.

Finally, Lobotomy [27] proposes to mitigate JIT spraying attacks by applying the principle of least-privilege to the Firefox JIT engine: by splitting the compiler and executor modules of the engine, indeed, it greatly reduces the amount of code that needs to access writable *and* executable pages. The main drawbacks of Lobotomy, with respect to Graffiti, are: 1) its overhead, that is sensibly high if compared with ours, and 2) the need to re-design the JIT engine of the protected process. The latter is particularly hindering because it greatly limits the portability of Lobotomy to other JIT engines. On the contrary, Graffiti can seamlessly protect *any program*, without modifying any of its inner components.

Data Spraying

Several defensive solutions have been proposed to avoid pivoting-based techniques [28, 33, 34]. One of the most deployed is part of EMET [28], a solution designed by Microsoft. EMET is a utility that helps to prevent vulnerabilities in software from being successfully exploited. Among other features, EMET also addresses the problem of stack pivoting attacks by checking if the stack pointer points outside of a process stack boundaries whenever a dangerous API is invoked. However, several researchers proved that it is possible to bypass the EMET technology in many ways [24, 18, 37] . The impact of these studies show that technologies that operate at the same level of execution of the malicious code need to be extensively tested and carefully designed to offer the desired protection and avoid possible bypasses. Consequently, these studies also shows the importance of designing reference monitors that operate at a lower level (e.g., at the hypervisor level) such as Graffiti to avoid these trivial attacks.

Moreover, Microsoft recently introduced two new countermeasures to hinder browser exploitation: isolated heap and delayed free [25, 45]. Both these techniques raise the bar for use-after-free attacks; as stated by the Fortinet Labs researchers [19], they also make

heap manipulation harder, but they are not a general solution as they protect only the Internet Explorer browser.

10 Conclusion

In this paper we propose an efficient and comprehensive solution to defeat spraying attacks by tracking the memory allocations of the system in an OS-independent way.

Overall, our paper makes several contributions: we introduce the concept of micro-virtualization that allows us to design an efficient and effective memory allocator tracker. We presented Graffiti, a general and extensible memory analysis framework that has good performance and it is freely available and open source. On top of it, we created three heuristics to detect and prevent spraying attacks. However, we believe that in the future Graffiti can also be extended and adopted in other domains, such as malware analysis or memory forensics.

References

[1] Alexa top domains. `http://www.alexa.com/topsites/category/`.

[2] Rop attack against data execution prevention technology, 2009. `http://www.h-online.com/security/news/item/Exploit-s-new-technology-trick-dodges-memory-protection-959253.html`.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[4] Greg Gagne Avi Silberschatz, Peter Baer Galvin. Operating system concepts. `http://os-book.com/`.

[5] Piotr Bania. Jit spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.

[6] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, June 2006.

[7] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[8] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394, May 2012.

[10] Liang Chen and Qidan He. Shooting the osx el capitan kernel like a sniper, 2016. `https://speakerdeck.com/flankerhqd/shooting-the-osx-el-capitan-kernel-like-a-sniper`.

[11] Ping Chen, Yi Fang, Bing Mao, and Li Xie. Jitdefender: A defense against jit spraying attacks. In Jan Camenisch, Simone Fischer-Hbner, Yuko Murayama, Armand Portmann, and Carlos Rieder, editors, *SEC*, volume 354 of *IFIP Advances in Information and Communication Technology*. Springer, 2011.

[12] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*, 2001.

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[14] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript, 2008.

[15] eEye Research. Microsoft internet information services remote buffer overflow (system level access), 2001. `https://web.archive.org/web/20061026101830/http://research.eeye.com/html/advisories/published/AD20010618.html`.

[16] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of*

the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '09, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, September 2010. https://code.google.com/p/hyperdbg/.

[18] Fireeye. Using emet to disable emet. https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disabl.html.

[19] Fortinet Labs. Is use-after-free exploitation dead? The new IE memory protector will tell you. http://blog.fortinet.com/.

[20] Ivan Fratric. Exploiting internet explorer 11 64-bit on windows 8.1 preview, 2013. https://ifsec.blogspot.com/2013/11/exploiting-internet-explorer-11-64-bit.html.

[21] Francesco Gadaleta, Yves Younan, and Wouter Joosen. Bubble: a Javascript engine level countermeasure against heap-spraying attacks. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *ESSoS, Pisa, 3-4 February 2010*. Springer Berlin / Heidelberg, January 2010.

[22] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 2–12, New York, NY, USA, 2006. ACM.

[23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A,3B,3C combined)*, March 2013.

[24] Bromium Labs. Bypassing emet 4.1. http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf.

[25] MWR Labs. Isolated heap & friends - object allocation hardening in web browsers. https://labs.mwrinfosecurity.com/blog/2014/06/20/isolated-heap-friends---object-allocation-hardening-in-web-browsers/.

[26] Lixin Li, James E. Just, and R. Sekar. Address-space randomization for windows systems. In *ACSAC*, pages 329–338. IEEE Computer Society, 2006.

[27] Jauernig Martin, Neugschwandtner Matthias, Milani-Comparetti Paolo, and Christian Platzer. Lobotomy: An Architecture for JIT Spraying Mitigation. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, September 2014.

[28] Microsoft. The enhanced mitigation experience toolkit. http://support.microsoft.com/kb/2458544.

[29] Microsoft. Structured exception handling overwrite protection (sehop). http://support.microsoft.com/kb/956607.

[30] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[31] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: automatically correcting memory errors with high probability. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 1–11. ACM, 2007.

[32] Michalis Polychronakis and Angelos D Keromytis. Rop payload detection using speculative code execution. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 58–65. IEEE, 2011.

[33] Aravind Prakash and Heng Yin. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 111–120, New York, NY, USA, 2015. ACM.

[34] Rui Qiao, Mingwei Zhang, and R. Sekar. A principled approach for rop defense. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 101–110, New York, NY, USA, 2015. ACM.

[35] Rapid 7. Metasploit penetration testing software. http://www.metasploit.com.

[36] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, August 2009.

[37] Duo Security. Wow64 and so can you bypassing emet with a single instruction. https://duo.com/assets/pdf/wow-64-and-so-can-you.pdf.

[38] Skylined. Microsoft internet explorer 6 - (iframe tag) buffer overflow exploit, 2004. https://www.exploit-db.com/exploits/612/.

[39] Skylined. Heap spraying high addresses in 32-bit chrome/firefox on 64-bit windows, 2016. http://blog.skylined.nl/20160622001.html.

[40] Kevin Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

[41] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[42] Alexander Sotirov. Heap feng shui in javascript, 2007.

[43] The PaX Team. Pax address space layout randomization. Technical report http://pax.grsecurity.net/docs/aslr.txt.

[44] Team Teso. 7350854.c, 2001. https://www.exploit-db.com/exploits/409/.

[45] Trendmicro Labs. Mitigating UAF Exploits with Delay Free for Internet Explorer. http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/.

[46] Vupen. Microsoft Internet Explorer javaprxy.dll COM Object Vulnerability / Exploit (Security Advisories). http://www.vupen.com/english/advisories/2005/0935.

[47] Vupen. Microsoft Internet Explorer "Msdds.dll" Remote Code Execution / Exploit (Security Advisories). http://www.vupen.com/english/advisories/2005/1450.