Kruiser: Semi-synchronized Non-blocking Concurrent Kernel Heap Buffer Overflow Monitoring

Donghai Tian[†*], Qiang Zeng[†], Dinghao Wu[†], Peng Liu[†]

[†]Penn State University
[*]Beijing Institute of Technology
{donghai, quz105, dinghao, pliu}@psu.edu

ABSTRACT

Kernel heap buffer overflow vulnerabilities have been exposed for decades, but there is still no practical countermeasure that can be applied to the OS kernel. Previous solutions either suffer from high performance overhead or compatibility problems with the existing kernel and hardware. In this paper, we present KRUISER, a concurrent kernel heap buffer overflow monitor. Unlike conventional methods, the security enforcement of which are usually inlined into the kernel execution, we introduce a concurrent monitor process, which decouples security mechanisms from the kernel's normal execution, leveraging the increasingly popular multicore architectures. To reduce the synchronization overhead between the monitor process and the running kernel, we design a novel semi-synchronized non-blocking monitoring algorithm, which enables an efficient runtime detection on live memory without incurring false positives. To prevent the monitor process from being tampered and provide guaranteed performance isolation, we utilize the virtualization technology to run the monitor in a trusted environment without affecting performance. We have implemented a prototype of KRUISER based on Linux and the Xen hypervisor. The evaluation shows that Kruiser can detect realistic kernel heap buffer overflow attacks effectively with minimal overhead. It imposes 2.7% throughput reduction on Apache and negligible performance overhead on SPEC CPU2006.

1. INTRODUCTION

Buffer overflows have been comprehensively studied for many years, but they remain as most severe vulnerabilities. According to the National Vulnerability Database, 319 buffer overflow vulnerabilities were reported in 2010, and 239 of them were marked as high severity [37].

Buffer overflows can be roughly divided into two categories: stack-based buffer overflows and heap-based buffer overflows. Both exist in not only user space but also kernel space. Compared with user-space buffer overflows, kernel-space buffer overflow vulnerabilities are more severe in that once such a vulnerability is exploited, attackers can override any protection mechanism. Recently, more and more realistic buffer overflow exploits have been released in modern operating systems including Linux [50], OpenBSD [53] and the latest Windows 7 system [33].

Many effective countermeasures against stack-based buffer overflows have been proposed, some of which, such as StackGuard [12] and ProPolice [24], have been widely deployed in compilers and commodity OSes. On the other hand, practical countermeasures against heap-based buffer overflows are few, especially in kernel space. To our knowledge, there are no practical mechanisms that have been widely deployed detecting kernel space heap buffer overflows. Previous methods suffer from two major limitations: (1) some of them perform detection before each buffer write operation [4, 25, 36, 26, 44], which inevitably introduce considerable performance overhead. This kind of inlined security enforcement can heavily delay the monitored process when the monitored operations become intense; (2) some approaches do not check heap buffer overflows until a buffer is deallocated [42, 3], so that the detection occasions entirely depend on the control flow, which may allows a long time for attackers to compromise the system. Other approaches [45, 14] either depend on special hardware or require the operating system to be ported to a new architecture, which are not practical for wide deployment.

In this paper, we present Kruiser, a concurrent kernel heap overflow monitoring system. Unlike previous solutions, Kruiser utilizes the commodity hardware to achieve highly efficient monitoring with minimal changes to the existing OS kernel. Our high-level idea is consistent with the canary checking methods, which first place canaries into heap buffers and then check their integrity. Once a canary is found to be tampered, an overflow is detected.

Different from conventional canary-based methods that are enforced by the kernel inline code, we make use of a separate process, which runs concurrently with the OS kernel to keep checking the canaries. To address the concurrency issues between the monitor process and OS kernel, we design an efficient data structure that is used to collect canary location information. Based on this data structure, we propose a novel semi-synchronized algorithm, by which the heap allocator does not need to be fully synchronized while the monitor process is able to check heap canaries continuously without being blocked. The monitor process is constantly checking kernel heap buffer overlows in an inifinite loop. We call this technique *kernel cruising*, and hence the name of our prototype, *Kruiser*. Our semi-synchronized cruising algorithm is non-blocking. The kernel execution is not blocked by monitoring, and monitoring is not blocked by the kernel execution. Thus the performance and other impacts on kernel execution characteristics are very small on a multicore architecture.

To further reduce the memroy and performance overhead, we have explored kernel heap management design properties to collect heap buffer region information at page level instead of individual buffers. This obervation enables us to design

a static fixed-size data structure. The normal approach is to maintain the collection of canary addresses of live buffers in a dynamic data structure, which requires hooking per buffer allocation and deallocation. Instead of interposing per heap buffer operation, we explore the characteristics of kernel heap management and hook the much less frequent operations that switch pages into and out of the heap page pool, which enables us to use a fix-sized static data structure to store the metadata describing all the canary locations. Compared to collecting canary locations in a dynamic data structure, we avoid the overhead of data structure growth and shrink; more importantly, it reduces overhead and complexity of the synchronization between the monitor process and the canary collecting code. This novel use of static data structures has low memory overhead, facilitates highly efficient monitoring, and reduces the maintenance overhead.

To prevent the monitor process from being compromised by attackers, we take advantage of virtualization to deploy the monitor process in a trusted execution environment. To achieve out-of-the-box monitoring,[1] one can run the monitor process in a trusted VM's user space and perform virtual machine introspection via VMM. However, frequent memory introspection introduces high performance overhead. To address this problem, Kruiser employs the Direct Memory Mapping technique, by which the monitor process can perform frequent memory introspection with only one-time involvement of the VMM, and thus reduces the performance overhead.

In summary, we make the following contributions:

- **Semi-synchronized concurrent monitoring:** We propose a novel non-blocking concurrent monitoring algorithm, in which neither the monotor process nor the monitored process needs to be fully synchronized to eliminate concurrency issues such as race conditions. We call this *semi-synchronized*.
- **Kernel cruising:** The novel cruising idea has been recently explored [59, 23]. It is nontrivial to apply this to kernel heap cruising.
- **Page-level buffer region vs. individual buffers:** We explore specific kernel heap management design properties to keep meta data at page level instead of at individual buffer level. This enables very efficient heap buffer metadata bookkeeping via a static fixed-size array instead of dynamic data structures and thus reduces the performance overhead dramatically.
- **Out-of-the-box monitoring via direct memory mapping:** To protect our monitor process, we apply direct memory mapping through virtualization to achieve out-of-the-box monitoring.

We have implemented a prototype of Kruiser based on Linux and the Xen hypervisor. To achieve the concurrent monitoring, we leverage the multiprocessor architecture that is very popular in recent commodity hardware. We evaluated the effectiveness of Kruiser by exploiting the heap buffer overflow vulnerabilities which are deliberately introduced by us. The experiment results show that Kruiser can detect kernel heap overflows effectively. In terms of performance and scalability, our kernel cruising approach is practical—it imposes negligible performance overhead on SPEC CPU2006, and the throughput slowdown on Apache is 2.7% in average.

[1]The other two options are to run the monitor process inside the same VM and inside VMM. In-the-box monitoring is not secure unless special treatment such as SIM [48] is enforced. Inside-VMM monitoring involves many VM-Exits, thus the overhead can be unacceptable.

2. CHALLENGES

In this section, we present the challenges we have encountered during the design and implementation of this work. Their solutions are presented in the next section.

C1. Synchronization. Since the monitor process checks heap memory which is shared and modified by other processes, synchronization is vital to ensure the monitor process locate and check live buffers reliably without incurring false positives.

Lock-based approach: A straightforward approach is to walk along the existing kernel data structures used to manage heap memory, which is commonly accessed in a lock-based manner. This requires the monitor process to follow the locking discipline. When the lock is held by the monitor process, other processes may be blocked. On the other hand, the monitor process needs to acquire the lock to proceed. Both the kernel performance and monitoring effect will be affected using the lock-based approach. Another approach is to collect canary addresses in a separate dynamic data structure such as a hash table. By hooking per buffer allocation and deallocation, the canary address is inserted into and removed from the hash table, respectively. However, it still does not reduce but migrate the lock contention, since the monitor process and other processes updating the hash table are synchronized using locks.

Lock-free approach: Scanning volatile memory regions without acquiring locks is hazardous [23], which usually needs to suspend the system to double check when an anomaly is detected. The whole system pause is not desirable and sometimes unacceptable. Another approach is to maintain the collection of canary addresses in a lock-free data structure. All processes update and access the data structure in a non-blocking manner. However, the experiments in our previous work [59], using the state-of-the-art extensible lock-free hash table [47] showed that the slowdown for each pair of buffer allocation and deallocation is more than 5X on average, although the scalability is much better than the lock-based counterpart. The reason is that the contention between accessing processes still leads to high overhead.

To achieve highly efficient concurrent monitoring, we designed a semi-synchronized algorithm which introduces zero-contention into kernel operations and performs non-blocking heap monitoring without incurring false positives or suspending the system.

C2. Self-protection. As a countermeasure against buffer overflow attacks, our component can become an attack target itself. We rely on a monitor process that keeps checking—that is, *cruising*—the kernel heap integrity. The busy process can be an explicit attack target. By killing the monitor process, attackers completely disable the detection. Attackers can also tamper or manipulate the data structure needed by our component to mislead or evade the detection. Thus we need to protect the safety of the monitor process and ensure the integrity of related data structures.

C3. Compatibility. Kernel heap management is among

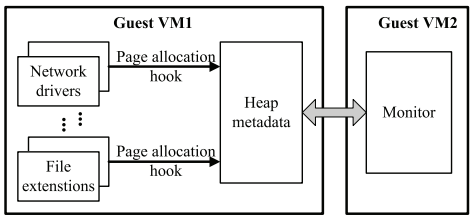

Figure 1: Overview of Kruiser.

the most important components in OS kernels, whose data structures and algorithms are generally well designed and implemented for efficiency. Thus, the concurrent heap monitoring should not introduce much modification for heap management. Moreover, the solution should be compatible with the existing systems including hardware.

3. OVERVIEW

Kruiser attaches one canary word at the end of each heap buffer and runs a separate monitor process, which keeps scanning, or *cruising*, the canaries to detect buffer overflows and runs concurrently with the monitored system. In this section we present an overview of the Kruiser architecture and the design choices addressing the challenges presented in the previous section. As shown in Figure 1, the monitor process is run in a separate VM as the monitored OS to strengthen self-protection. The heap buffer metadata is kept in the monitored VM to achieve efficient updating. The monitor cruises over the heap metadata via an efficient technique called direct memory mapping. Once a kernel heap buffer canary is found corrupted, an overflow is reported. In this architecture, the monitor process needs to retrieve the canaries reliably, while the monitored system may be deallocating the buffers and heap pages.

To address this synchronization challenge (C1), without imposing high overhead, we explore the characteristics of kernel heap management, and propose to interpose heap page allocation and deallocation, through which we maintain concise metadata describing canary locations in a separate efficient data structure. Compared to interpose per buffer allocation and deallocation, the interposition is lightweight and the resultant overhead is much lower. The per page metadata is concise, which enables us to use a fix-sized static data structure to store it. Compared to using a concurrent dynamic data structure to collect canary addresses, the contention due to synchronizing data structure growth and shrink and the overhead due to data structure maintenance (node allocation and deallocation) are completely eliminated. More importantly, as the monitor process traverses our own data structure rather than relying on existing kernel data structures, it is more flexible to design the synchronization algorithm, i.e. the monitor process do not need to follow the synchronization discipline imposed by the kernel data structure. Therefore, we are able to design a highly efficient semi-synchronized non-blocking algorithm, which enables the monitor process to constantly check the live memory of the monitored kernel without incurring false positives.

To address the self-protection challenge (C2), we apply the virtualization technology to deploy the monitor process into a trusted environment (Figure 1). To ensure the same efficiency as in-the-box monitoring, we introduce the Direct Memory Mapping (DMM) technique, which allows the monitor process to access the monitored OS memory efficiently. To protect our data structure from being overflowed or underflowed, we apply two write-protected pages surrounding our data structure. More comprehensive protection mechanisms are presented in Section 7.

To address the compatibility challenges (C3), we make little change to the existing kernel heap management, relying on the commodity hardware. Specifically, we hook the allocation/deallocation that adds/removes pages into/from the heap page pool to update the corresponding heap metadata in our data structure, for which we make use of the existing page allocator to allocate kernel pages beforehand. On the other hand, the monitor component is located in another protection domain, which utilize the DMM technique to cruise over the kernel heap by looking up the metadata.

4. KERNEL CRUISING

In this section, we present the semi-synchronized non-blocking kernel cruising algorithm. We introduce the data structure used in the algorithm in Section 4.1. We discuss potential race conditions in Section 4.2 and describe our algorithm in Section 4.3.

4.1 Page Identity Array

Kernels usually maintain heap metadata in dynamic data structures. For example, Linux kernel uses a set of lock-based lists to describe the heap page pool. It is tempting to walk along the existing data structures to check heap buffers. This way the concurrent monitor process has to follow the locking discipline, which would introduce intense lock contention. Another concurrent approach, as used in kernel memory mapping and data analysis for kernel integrity checking [23], is to check without acquiring locks and freeze the monitored VM for double-check to avoid false positives, which may require suspending the VM frequently.

Instead of relying on kernel-specific data structures, we maintain a separate structure called Page Identity Array (PIA). Its basic form is a static array data structure with each entry recording the *identity* of a page frame. A variety of page identity information can be of interest, such as per page signature, access control, accounting and auditing data. With regard to concurrent heap monitoring, a PIA entry records whether a page frame is used for heap memory, and if so, the metadata that is used to locate canaries within the page. The first entry corresponds the first page frame, and so forth. Since the kernel memory address space is fixed, the size of PIA structure can be pre-determined. This way we only need to hook functions that add pages into the heap page pool and that remove pages from it, updating metadata in the corresponding entries. The monitor traverses the PIA structure and check canaries according to the stored metadata. Compared to interposing per buffer allocation and deallocation and collecting canary addresses in a dynamic data structure, the overhead due to function hooking and data structure maintenance is largely reduced. We postpone details about metadata and memory overhead analysis in Section 5.

The idea of using a fixed-size data structure is due to the insight into kernel heap management. We assume that a kernel page, if used for heap memory, is divided into buffer objects of equal size and that all the buffers in this page are arranged as an array, which is true in most commodity

systems, such as Linux, Solaris, and FreeBSD. Given a heap page and its initial buffer object address and size, the monitor process can locate all the buffers within this page, such that the metadata stored in each PIA entry can be small. Before a process (or a kernel thread)[2] adds a page into the heap page pool, the canaries within the page are initialized and the corresponding PIA entry is updated. By scanning the canaries within each page, the monitor process detects buffer overflows. Although some buffer objects are not allocated and some canary checking may be not necessary, the simple read operations do not introduce much overhead. For 64-bit systems with large address space and physical memory, however, the flat PIA structure is not scalable enough and sparse kernel heap pages lead to a concern of significant ineffective scanning. In such as, a multi-level PIA similar to page table can be used.

4.2 Race conditions

Exploring the characteristics of kernel heap management, we proposed the static PIA structure, which mitigates heap monitoring from kernel-specific heap data structure accesses and supports highly efficient random access. Nevertheless, synchronization between the monitor process and processes updating page identities is still an issue. For example, when the monitor process reads an entry, another process may be updating it. Without synchronization, the consistency of PIA entries cannot be ensured, which implies the monitor process cannot retrieve heap buffers reliably.

Before we present the kernel heapcruising algorithm, we first discuss the potential race conditions for sharing the PIA structure, which motivate our semi-synchronized design in Section 4.3. Three categories of processes need to access the PIA structure: the monitor process, processes updating PIA entries when pages are added into and removed from the pool, respectively. When multiple processes access the PIA structure, a variety of race conditions can occur, some of which are subtle.

Non-atomic entry write: As the update of a PIA entry is not atomic, a race condition occurs if we allow multiple processes to update the same entry simultaneously, which would corrupt the entry. Lock-based synchronization is simple, but it incurs high performance overhead and blocks heap operations.

Non-atomic entry read: When the monitor process is reading a PIA entry, another process may be updating it. However, as the read and update of an entry are not atomic, the monitor process may read inconsistent entry value.

Time of check to time of use (TOCTTOU): For each entry if the corresponding page is in the heap pool, the monitor process checks canaries within that page, during which, however, the page may be removed from the pool and used for other purposes, such that false alarms may be issued.

To avoid false alarms, it is tempting to double check whether the page has been removed from the heap page pool when a canary is detected tampered. Specifically, a flag field indicating whether the page is in the pool is contained in each entry. A process removing the page out of the heap page pool resets the flag; when a heap buffer corruption is detected, the monitor process double checks the flag to make sure the page is still in the pool. A buffer overflow is reported only when a canary is tampered and the flag in the PIA entry is not reset. However, it cannot avoid the ABA hazard as discussed below.

ABA hazard: An ABA hazard occurs when one process reads a value A from some position, and then needs to make sure the position is not modified since last read by reading it again and comparing the second read value with A. However, between the two reads, other processes may have updated the position from value A to B then back to A. In our case, it may lead to an ABA hazard if the monitor process intend to determine whether the entry has been updated by reading the flag twice, considering that other processes may remove the page from the heap page pool and then add it back between the two reads, such that the idea of double-checking the flag can still lead to false alarms due to ABA hazards.

Compared to the idea of walking along existing kernel data structures, we apparently have conquered nothing except migrating the synchronization problems to the PIA structure. However, as presented below, we propose a semi-synchronized algorithm to resolve all the problems without incurring false positives or high synchronization overhead.

4.3 Semi-synchronized Non-blocking Cruising

We propose an efficient semi-synchronized non-blocking kernel cruising algorithm, as shown in Figure 2, that works with the PIA structure. It resolves the concerns of race conditions without introducing complex synchronization mechanisms, such as fine-grained locks and intricate lock-free data structures.

We add an unsigned integer field version in each entry, which records the "version" of the corresponding page. It is initialized to be an even number when the corresponding page is not in the heap page pool. Whenever a page is added into or removed from the pool, its corresponding version number is incremented by one, so that an odd version number indicates a heap page, and an even number indicates a non-heap page. Because the size of the version field is one word, the read and write of a version value is atomic, which is critical for the correctness of our algorithm.

Avoid Concurrent Entry Updates: The kernel commonly has its own synchronization mechanisms to prevent one page frame from being manipulated for inconsistent purposes at the same time. For example, Linux function kmem_getpages and kmem_freepages, which add page frames into and remove them from the heap page pool, respectively, operate on page frame in a critical section with lock protection. These two functions correspond to AddPage and RemovePage in Figure 2, respectively. The PIA entry update operations can be put into the critical section of these two functions; it is thus ensured that two processes cannot update the same entry simultaneously. By leveraging the existing synchronization mechanisms in kernel to maintain the PIA entries, the additional overhead is minimal since updateing metadata in a PIA entry is fast. As long as the kernel prevents one page frame from being manipulated by two processes simultaneously, there should be synchronization mechanisms serving for this purpose, so the "free-ride" is widely available.

Avoid Using Inconsistent Entry Value: Instead of preventing the monitor process from reading inconsistent entry value, we allow it to occur. However, we use a double-check algorithm to detect potential inconsistency and avoid using inconsistent values. We read the version field in an

[2]In this paper we will use the two terms interchangeably.

```
1 //Add a page into the heap page pool
2 AddPage(page){
3     ...
4     /* Inside critical section */
5     Initialize all the canaries within the page
6     Update the metadata in PIA[page];
7     smp_wmb(); // This write memory barrier enforces a
          store ordering
8     PIA[page].version++;
9     ...
10 }
11
12 //Remove a page out of the heap page pool
13 RemovePage(page){
14     ...
15     /* Inside critical section */
16     for (each canary within the page)
17         if (the canary is tampered)
18             alarm(); // A Buffer overflow is detected
19     PIA[page].version++;
20     ...
21 }
22
23 Monitor(){
24     uint ver1, ver2;
25     for (int page = 0; page < ENTRY_NUMBER; page++)
           {
26         ver1 = PIA[page].version;
27         if (!(ver1 % 2))
28             continue; // Bypass non−heap page
29
30         smp_rmb(); // This read memory barrier enforces a
               load ordering
31         Read the metadata stored in PIA[page];
32         smp_rmb();
33         ver2 = PIA[page].version;
34         if (ver1 != ver2)
35             continue; // Metadata was updated during the
                   read
36
37         for (each canary within the page){
38             if (the canary is tampered)
39                 DoubleCheckOnTamper(page, ver1);
40         }
41     }
42 }
43
44 DoubleCheckOnTamper(page, ver){
45     uint ver_recheck = PIA[page].version;
46     if (ver_recheck != ver)
47         return; // The page was already removed/reused
48     alarm(); // A buffer overflow is detected
49 }
```

Figure 2: Kruiser monitoring algorithm.

entry first (Line 26), and then retrieve other entry fields followed by another read of the version field (Line 33). The page is to be scanned if and only if the two reads of the version field retrieve identical odd version numbers. Here we assume the wraparound of the version value does not occur between the two reads. Considering that page frame switch in and out of the kernel heap pool is infrequent, it very unlikely that the version number wraps around a 32-bit unsigned integer between the two reads.

Specifically, assume there is a non-heap page frame and the AddPage function adds it into the heap page pool. In its critical section it first updates the metadata and then the version number (Line 8) in the corresponding page entry, such that if the monitor process reads the version number of the entry being updated and the read is before the version number update (Line 8), it will retrieve an even number, which indicate a non-heap page. The monitor process will bypass this page (Line 27) according to our algorithm. A write memory barrier (Line 7) is inserted before the version number update, which preserves an observable update order. It is a convention to assume a sequential consistency memory model in the parallel computing literature when describing a concurrent algorithm; however, the observable update sequence [35] is vital to the correctness of our algorithm, so we point it out explicitly.

The version number is not incremented until RemovePage removes the page from the pool. It does not need write memory barriers around the version update because the enter and exit of a critical section imply a full memory barrier, respectively. Therefore, as long as the two reads of the version field retrieves identical odd values, the retrieved metadata values are consistent. Two read memory barriers (Line 30 and 32) are inserted into the Monitor function, such that an observable load ordering is enforced among the reads of the version number and metadata. But note that the read and write memory barriers are not needed on x86 and AMD64 platforms [34], as they already preserve the loads and stores orders we need.

Identify TOCTTOU and ABA Hazards: Without locks or other synchronization primitives, it is difficult to avoid TOCTTOU and ABA hazards. Rather than avoiding the hazards, the algorithm takes a different approach to recognizing potential hazards to avoid false alarms. When a canary is found changed, the monitor process does not report an overflow immediately. Instead, it makes sure the page being checked has not ever been removed out, which is indicated by the version number again. As long as the version number does not change compared to the last read (Line 46), it can be determined that the page has persisted as a heap page; in this situation, if a canary is found corrupted, a buffer overflow is reported without concerns of false positives.

The non-blocking algorithm is constructed using simple reads, writes, and memory barriers without introducing complicated and expensive synchronization mechanisms. The monitoring is *wait-free* as it guarantees progress in a finite steps of its own execution; i.e., it is non-blocking. The monitor process reads version numbers to determine its control flow, so it is lightly synchronized, while other processes manipulating heap pages make progress without being synchronized or blocked by the monitor process. In other words, the synchronization is one-way. That is why we call it a *semi-synchronized non-blocking cruising*. It is semi-synchronized in anther sense. On PIA entries, write-write is synchronized with a free-ride from the existing kernel functions, while read-write is not synchronized. It resolves the concern of a variety of subtle race conditions without the need of freezing the entire system for recheck. It does not have false positives and enables efficient concurrent heap monitoring.

5. SYSTEM IMPLEMENTATION

5.1 Background

We first describe the *slab allocator* schema[3] used in Linux for kernel heap management. The slab allocator uses cache

[3]The similar schema is also widely used in other commodity systems, such as Solaris and FreeBSD

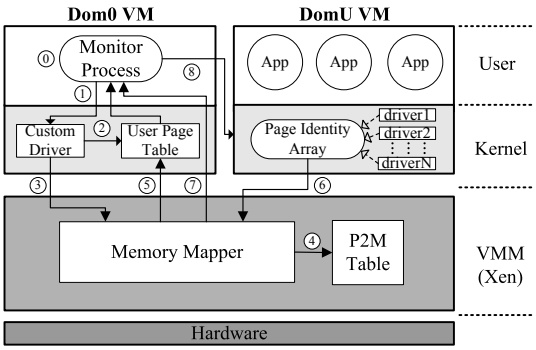

Figure 3: Kruiser Architecture. The numbers in the small circle indicate Kruiser's work flow.

to organize and manage heap buffer objects. Each cache is a "store" of objects of the same type (specific caches), e.g. the cache for task_struct type, or the same size (generic caches, where kmalloc allocates objects).

A cache is divided into slabs; each slab consists of one or more contiguous page frames and arranges buffer objects of the same size in an array. The metadata (slab and object descriptors) of a slab, which is used to describe the object arrangement and status (allocated or deallocated) within the slab, can be stored in or out of the slab. Slabs are linked into lists; a cache grows by allocating new slabs and adding them into its slab lists.

5.2 Architecture

We developed a prototype of Kruiser based on 32-bit Linux and the Xen hypervisor. As Figure 3 shows, the architecture can be divided into three parts: VMM, Dom0 VM, and DomU VM (the monitored VM). Dom0 VM contains the monitor process and the custom driver, which reside in user space and kernel space, respectively. The custom driver is used to help the monitor process release memory with its page tables retained. A tiny component, namely Memory Mapper, inside the VMM is used to map the kernel memory of the monitored VM to the page table entries retained above. The Page Identity Array and the interposition code reside in the the kernel space of DomU VM.

5.3 Direct Memory Mapping

To achieve out-of-the-box monitoring, a common method is to run a monitor process in a trusted VM's user space and perform virtual machine introspection (VMI) via the underlying VMM. However, frequent memory introspection would incur high performance overhead. Each such operation requires VMM to walk the target VM's page table and map the machine frame numbers (MFN) to the monitor process. To address this problem, we introduce Direct Memory Mapping (DMM), by which the monitor process can perform frequent memory introspection with only one-time involvement of the VMM. The basic idea is to let the VMM manipulate the page table of the monitor process such that the user-space monitor can access the kernel memory of the target OS directly. Figure 4 illustrates the DMM architecture. Basically, the working process of DMM can be divided into three stages.

First, the Monitor Process allocates a chunk of memory whose size is determined by the number of pages that are supposed to be used by the kernel heap (⓪ in 3). For

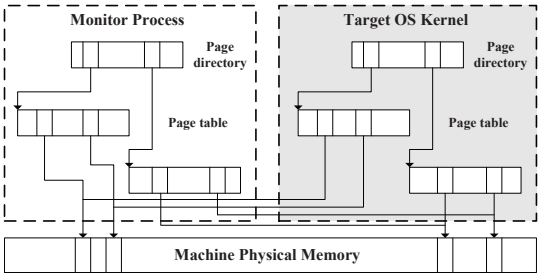

Figure 4: The direct memory mapping mechanism.

Linux, the kernel heap only resides in the pages that are directly mapped by the OS kernel. In a 32-bit operating system, the maximum size is 896MB even if the machine physical memory size is bigger than 896MB. The goal of this stage is to create a contiguous range of virtual addresses. By properly manipulating their corresponding page table entries (PTEs), the VMM can allow the monitor process using its private virtual addresses to access the memory of the target OS kernel. However, due to the demand paging mechanism adopted by Linux, the PTEs are not actually established when the virtual addresses are created. Therefore, we need to access all the created memory area before operating on these PTEs.

Second, the Monitor Process notifies the Custom Driver to recycle the newly allocated pages (①), but keeping the PTEs unchanged. This step is necessary because the Monitor Process only need the new virtual addresses but does not use the associated allocated pages; getting these pages back to the trusted VM will save a lot of memory. For this purpose, we take four steps: 1) The Custom Driver first walks the page table of the Monitor Process to get the PTEs according to the virtual addresses that belonge the Monitor Process (②). 2) Then, with these identified PTEs, the Custom Driver finds the associated page descriptors that are used for the page frame management by the OS. 3) After that, the Custom Driver clears the relevant flags of these page descriptors (e.g., active flag), and resets their reference counters, map counters as well as other related information. 4) Finally, the Custom Driver invokes the API of the buddy system (i.e., __free_page()) to release the page frames.

Third, after the Custom Driver finishes recycling pages, it informs the Memory Mapper to perform DMM for the Monitor Process (③). By looking up the DomU's physical-to-machine (P2M) table (④), the Memory can get all the MFNs of the DomU. With these mapping information needed, the Memory Mapper updates the PTEs of the Monitor Process accordingly. More specifically, given the newly created virtual address, the Memory Mapper walks the User Page Table to find the corresponding PTEs (⑤). Then it changes their page frame numbers to the associated MFNs that are found in the P2M table. By doing so, the Monitor Process can traverse the entire kernel of the target OS with its own page table. In other words, when the Monitor Process uses its new private virtual addresses, it can read/write the memory of the target OS directly.

Once the Page Identity Array is allocated, it invokes a hypercall to notify the underlying VMM (⑥), which then informs the monitor process to begin cruising over the kernel heap (⑦)(⑧), relying on the PIA.

Reducing TLB Miss. Considering the big range of mem-

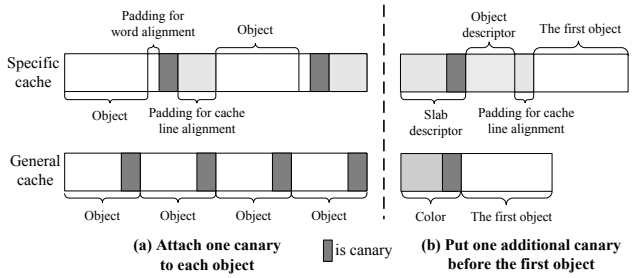

Figure 5: Inserting canaries into kernel objects.

ory area that the Monitor Process may need to access, the kernel cruising will incur great TLB miss when a large number of kernel slabs are produced. To address this problem, we exploit the extended paging mechanism that is supported by commodity microprocessors. Specifically, we set the Page Size flag in the page directory entries, enabling the size of page frames to be 2MB instead of 4KB (the page frame will be 4MB in size if it is in None-PAE mode). It is worth mentioning that we also need the hypervisor to support the extended paging. Fortunately, Xen (with PAE enabled) mainly uses 2MB super pages to allocate memory for guest VMs. On the other hand, to ensure the extended paging working properly, we require the starting virtual address allocated for the monitor process should be 2MB-aligned. To meet this requirement, the Monitor Process needs to allocate 2MB extra memory during the first stage, and then adjust the starting virtual address to be 2MB-aligned before performing DMM.

5.4 Placing canaries

To detect underflows as well as overflows, it is straightforward to place two canaries surrounding each buffer. However, since kernel threads usually assume the allocated buffer is cache line aligned, by reserving a word for the canary at the beginning of each buffer, the assumption is broken, which may stress the TLB and impose considerable performance overhead as all dynamic kernel objects are not cache line aligned. The scheme is adopted in Linux when the slab debugging option is enabled, and We exploit the property of buffer objects within each slab, which are arranged as one array, such that even we put only one single canary at the end of each buffer, we can still detect underflows: the underflow of one object will corrupt the canary of its preceding object. This way the cache line alignment is not changed.

As shown in Figure 5(a), the placement of canaries in objects for specific caches and general caches are a little different. For objects in specific caches, each of which is used for a specific data type, the object size is fixed. We can place the canary right after the (word-aligned) object. The cache line alignment of the next object is not affected. For objects in general caches, the real object may be smaller than the buffer, so the canary is put at the end of each buffer. When kmalloc is invoked, the requested buffer size is incremented by one word to accommodate the canary.

Although the scheme above works well to detect underflows (and overflows), but can not deal with underflows occurred in the first object, as there is no canary preceding it. To tackle this issue, as shown in Figure 5(b), we exploit the existing infrastructure to add a canary before the first object. Specifically, if the slab descriptor is located rightly before the first object, the canary placed at the end of the slab descriptor buffer(, note that the slab descriptor itself is allocated from specific caches) suffices; or if there is a slab color,[4] we put a canary in the last word of this color.

```
1 struct PIA_entry{
2     unsigned int version;
3     short mem; // the starting address of the first object
4     short slab_size; // the size of the slab descriptor
5     int obj_size; // the actual size used by each object
6     int buffer_size; // the whole size for each object
7     int number; // the number of objects in this slab
8 };
```

Figure 6: PIA entry.

The canary value is the XOR result of the buffer address and a secret key, such that even a canary is leaked due overread bugs [52], it is difficult to guess other canaries without the buffer addresses. It is intended to defend canary guessing attackers before the system is compromised. We will present a scheme in Section 7.1 which prevent attackers from guessing a canary even the system is controlled by attackers.

5.5 Locating canaries

After inserting canaries around kernel buffers, the next step is to locate and check these canaries for the Monitor Process. For this purpose, we hook the slab allocations and de-allocations to store the metadata into the PIA entries, by which the monitor process can get all the canary locations in the kernel heap.

The PIA entry consists of several fields, which is shown in Figure 6. The mem field record the starting address of the first object within the slab. As each PIA entry corresponds to one physical page, we only need to remember the last 12 bit of the address, which equals the offset within one page. For the obj_size field, we store the actual object size, including the size of padding for word alignment.

By adding the starting address of one object and its actual object size, we can get the canary address. To acquire the starting address of the next object, the PIA entry contains the buffer_size field, which refers to the whole object size after adding the canary as well as the padding for cache line alignment. The num field indicates the number of objects within a slab. To locate the canary that resides in the slab descriptor, we record the slab descriptor size in the slab_size field, which additionally includes the size of the object descriptor and the following padding. With the starting address of the first object subtracting the slab descriptor size, we get the starting address of the slab descriptor and then locate the canary, whose offset within the slab descriptor is pre-determined. On the other hand, if the slab descriptor is kept off the slab, we set the value of the slab_size to zero. Accordingly, we employ a different method to locate the canary before the first object. In particular, we check whether the starting address of the first object is page-aligned, if not, it indicates there is a color placed in the front. Then, we can check the canary safely.

As introduced previously, kernel heap are managed in different slabs, one of which consists of one or more physically contiguous pages. Therefore, the slab that contains several pages should correspond to several entries in the PIA. In order to facilitate recording the slab canary information into PIA entries, we just use the first associated entry to store the whole information, and keep other associated entries empty.

[4] A slab color is a padding put in the beginning of each slab to optimize the hardware cache performance.

It is worth mentioning that we utilize the page allocator to dynamically allocate kernel memory for the PIA data structure during the kernel's initialization. Basically, the total memory occupied by the PIA is determined by the number of pages in the heap. However, the proportion is unchanged even if all the physical memory are used by the kernel heap. Since each PIA entry has only 20 bytes in our implementation, the memory overhead is as low as 20/4096.

Since the PIA data structure and kernel heap reside in the same memory area, the PIA may also become the overflow target. To address this problem, we allocate two guard pages surrounding the PIA, and then leverage the shadow page table (SPT) management subsystem in the Xen hypervisor to set them write-protected. In this way, the hypervisor can trap the event whenever attackers attempt to overwrite through the guard page. However, this method is still limited in defending against advanced attacks (e.g., manipulating the PIA data structure directly). For more comprehensive solution, we discuss it further in Section 7.

6. EVALUATION

To evaluate Kruiser, we conducted effectiveness tests and measured performance overhead. All the experiments were ran on a Dell Precision Workstation with two 2.26GHz Intel Xeon quad-core processors and 6GB memory. The Xen hypervisor (with PAE enabled) version is 3.4.2. We used Ubuntu 8.04 (linux-2.6.24 with PAE enabled) as Dom0 system and Ubuntu 8.04 (linux-2.6.24 with PAE disabled) as DomU system (with HVM mode). Moreover, we allocated 1 GB memory and 4 VCPU for this DomU system.

6.1 Effectiveness

To test whether Kruiser can detect heap buffer overflows, we deliberately introduced three explicit vulnerabilities [43, 50] in the Linux kernel, and then exploited these bugs. In our first test, we modified the kernel function cmsghdr_from_user_compat_to_kern, making it process some user-land data without sanitization. By doing so, malicious users could launch heap-based buffer overflow attacks via the sendmsg system call. For the second test, we loaded a vulnerable kernel module that is developed by ourselves. The function of this module is to use a dynamic general buffer to store certain data transferred from the user-land. However, the module does not perform boundary check when it stores the user data. In the third test, we also employed a loadable kernel module to export a bug in kernel space. Unlike the second test, we constructed a specific slab in this module, and allocated the last object in this slab to store certain user-land information [50]. As a result, this vulnerability enables attackers to overwrite a page next to the slab by transferring large size data into the kernel object. We then launched three types of heap-based buffer overflow attacks, respectively. Each attack was executed 10 times and Kruiser detected all these overflows successfully. The experiment results indicate that Kruiser is effective in defending against kernel heap buffer overflow attacks.

6.2 Performance Overhead

To evaluate the performance overhead, we carried out a set of experiments. First, we executed the micro-benchmark to measure the overhead at the kernel function call level. Then, we ran the SPEC CPU2006 Integer benchmark to test the application-level overhead. Each of these experiments was conducted in three different environments, including original Linux, Linux with slab debug enabled (referred as Linux-debug subsequently), and Kruiser.

Table 1: The average normalized execution time of kernel APIs with Kruiser and Linux-debug when compared with original Linux.

Kernel APIs	Debug	Kruiser
kmem_cache_create	1.13	1.08
kmem_cache_alloc(1st)	5.05	1.19
kmem_cache_alloc(2nd)	23.62	1.06
kmem_cache_free	5.25	1.06
kmem_cache_destroy	1.19	1.10

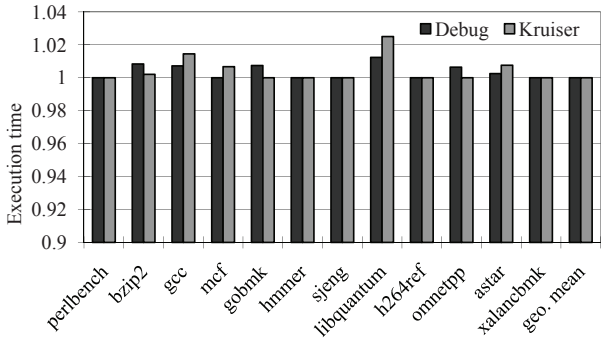

Figure 7: SPEC CPU2006 performance (normalized to the execution time of original Linux).

Micro-Benchmark. To evaluate the performance of the APIs exported by the slab allocator, we implemented a kernel module that invokes the APIs to allocate specific kernel objects with varied bytes (from 20 to 400 bytes). Table 1 shows the average execution time of Kruiser and Linux-debug, which are normalized by the execution time of original Linux. For Kruiser, we can see that there are two kernel APIs with 6% and the other three with less than 20% performance overhead. However, the overhead introduced by Linux-debug is very significant, especially for kmem_cache_alloc and kmem_cache_free. This is reasonable because we only add some small code to slab constructions and functions kmem_cache_create and kmem_cache_destroy while Linux-debug performs lots of extra work (e.g., checking the canaries) during object allocations and deallocations.

Application Benchmark. For application-level measurement, we used SPEC CPU2006. Figure 7 shows that the average performance overhead for both Kruiser and Linux-debug are negligible. However, we also notice that in some test cases the performance of Kruiser is not as good as Linux-debug. The main reason is that the performance of Linux-debug depends on the number of kernel object allocations while some applications like libquantum in SPEC only trigger very few kernel buffer allocations. As a result, the enforcement code inlined in Linux-debug almost does not get executed during the tests. On the other hand, Kruiser keeps checking all the kernel objects including the ones allocated before the test cases, which inevitably results in some performance overhead even if there isn't any kernel object allocation.

6.3 Scalability

We tested the throughput of the Apache web server with

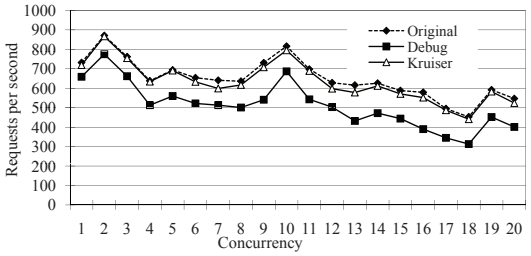

Figure 8: Throughput of the Apache web server for varying numbers of concurrent requests.

concurrent requests. In this test, we ran Apache 2.2.8 to serve a 3.7KB html web page. We used ApacheBench 2.3 on another machine—a Dell PowerEdge T300 Server with a 1.86G Intel E6305 CPU, 4 GB memory and Ubuntu 8.04 (linux-2.6.24)—to measure the Apache throughput over a GB LAN network. Each time we issued 10k http requests with various numbers of concurrent clients. We observed that the number of the kernel object allocations that are related to Apache increases along with the concurrency level. As shown in Figure 8, the relative performance overhead imposed by Linux-debug is increased when we add more concurrent http clients. On the contrary, the relative performance overhead introduced by Kruiser is almost unaffected by the concurrency level. Moreover, Kruiser only incurs about 2.7% performance degradation while Linux-debug imposes more than 22% performance slowdown on average.

6.4 Detection Latency

The kernel heap overflows happen when an attacker overwrites a kernel buffer to alter the content of its adjacent memory. However, the attacker cannot achieve the exploit until content of the overwritten memory gets executed or used in a malicious manner. Therefore, if the detection latency (the average time consumed to scan the whole PIA once) of Kruiser is short enough, Kruiser is able to detect the heap overflow before the exploit is achieved.

To evaluate the detection latency, we recorded the average cruising cycles (i.e., the average time for scanning all the PIA entries) for different applications in SPEC CPU2006. As shown in Table 2, 10 of 12 applications' average cruising cycles are shorter than 6.6 ms, and the other two applications' are below 7.3 ms. We also recorded the number of scanned kernel objects in each cruising cycle. The results indicate that the average cruising cycle is mainly determined by the average number of scanned kernel objects. Let N be the number of scanned kernel objects and T the average time for the monitor process to check a kernel object. We have $C = NT$, where C is the cruising cycle. We can reduce the cruising cycle by keeping N small. One approach is to divide the PIA entries into different parts, and for each part, we create a separate monitor process. Another approach is to only monitor the general buffers in the kernel space, excluding other kernel objects. This is pratical because attackers mainly exploit the kernel general buffers in the real world.

Table 2: Different cruising cycle for different applications in the SPEC CPU2006 benchmark (The cruising number refers to the number of kernel objects that are scanned in each cruising cycle).

Benchmark	Maximum cruising number	Minimum cruising number	Average cruising number	Average cruising cycle(μs)
perlbench	109,514	103,921	107,274	7,271
bzip2	79,542	75,279	75,754	6,327
gcc	77,992	76,581	77,334	6,372
mcf	81,024	77,623	77,812	6,456
gobmk	79,095	78,576	78,854	6,421
hmmer	79,140	78,730	78,816	6,374
sjeng	79,693	79,044	79,121	6,654
libquantum	80,358	79,387	79,605	6,396
h264ref	79,766	79,383	79,579	6,390
omnetpp	80,887	80,097	80,246	6,524
astar	98,427	81,785	87,550	6,589
xalancbmk	100,481	99,517	99,909	6,915

7. DISCUSSION

7.1 Guaranteed Detection

Kruiser races with attackers: as long as an exploit cannot succeed within a cruise cycle after a canary is corrupted, it is bound to be prevented. In addition, even an attacker has compromised the system by exploiting an kernel heap buffer overflow vulnerability and enabled a remote shell with root privileges, the canary corrupt should be detected before the attacker keys in the first command, since a cruise cycle is normally less than 10 milliseconds. In this sense, Kruiser "raised a bar" for attackers.

However, automatic attack vectors such as worms can be fast and advanced attacks may directly manipulate our data structures or try to recover the corrupted canaries using the keys. Moving the data structures and keys to a separate VM gains security but can lead to high performance overhead. Here we present a scheme that prevents attackers from recovering the corrupted canary, even after the system has been compromised and entirely controlled by attackers. The scheme combines two techniques. One is adapted from Secure-In-Vm (SIM) monitoring introduced by Sharif et al. [48], and the other is Signed Canary.

SIM proposes to put the monitor back into the VM and provide a one-way memory view to the monitor, such that the monitor code can access the whole address space, while the monitored kernel cannot access some memory regions reserved by the monitor. The discriminative memory views are enabled by the underlying VMM via page table manipulation. As shown in Figure 9, we can put the data structure and keys in the reserved memory regions, such that attackers can not access them directly. SIM allows the monitor to expose some interfaces. We can move the code from the critical section of AddPage and RemovePage in Figure 2 to the reserved memory regions and expose two interfaces to invoke them, respectively. One more check is added in the protected code to to prevent from duplicate page adding, such that attackers cannot recover canaries by invoking duplicate AddPage. On the other hand, if attackers invokes RemovePage, the final round of canary checking in the protected code can detect overflows. Different form SIM that enforces inlined security enforcement, our monitor process still runs out of the monitored VM.

Current canary generation algorithm is simple (using XOR). Capable attackers may infer the value of a canary by, for example, reading its neighboring canaries, thus recover the corrupted one. Based on approaches such as SIM to pro-

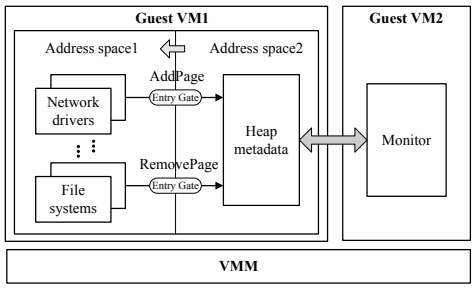

Figure 9: Strengthened Kruiser.

tecting the keys, we propose Signed Canary, which prevents attackers from inferring canaries by enhancing the canary generation algorithm using applied cryptography. Cryptography algorithms for MAC or symmetric encryption can be used for this purpose. The unique memory address of each canary is cryptographically hashed or encrypted using secret keys to generate the canary value, such that it is difficult for attackers to infer a canary without the key. Instead of using a single key, a group of keys can be used to enhance the algorithm: the nth canary in a page frame can be generated using the mth key in the group, and all the keys and the mapping of key usage are initialized when the guest OS is booted. To accelerate the hash calculation, a 64K-entry lookup table for 16-bit values substitution (hash) can be used, so that a canary can be generated with two lookups. Our canary generation requires only milliseconds of resistance, as all buffer overflows can be detected within one cruising cycle, which implies that the cryptography for our purpose can be potentially simple and efficient.

With the guaranteed detection, attackers can not hide their attacks and are bound to be detected within milliseconds after compromising the system, unless they know the exact canary to be corrupted beforehand. Combined with the checkpoints technique, this enables a system to recover the nearest clean state.

7.2 Viable Deployment

Large data centers using shipping-containers packed with thousands of servers each are not uncommon nowadays. Therefore, scalable deployment is a critical requirement for intrusion detection measures in data centers.

Unlike traditional interposition-based monitors, which may intervene normal functionalities frequently, Kruiser imposes minimal interference and performs monitoring in parallel with the monitored VM. In addition, the performance isolation provided by the underlying VMM ensures the two entities do not abuse computing resources to interfere with each other, which is a desirable property for both data center administrators and consumers.

According to our evaluation, Kruiser can normally perform a cruising cycle in milliseconds. Assume the acceptable intrusion detection latency is one second, a single Kruiser instance is then eligible for monitoring dozens of VMs on a physical machine. It can alternatively perform a discriminative scanning based on the importance of services running on different VMs.

With the popularity of multicore architectures, servers built with many cores are more and more common. The hardware evolution trend embraces the concurrent monitoring fashion, as the unit cost for a monitor instance decreases sharply. The cost will be very low if not negligible for running additional Kruiser instances on separate cores. Therefore, the scalability and low cost properties imply that Kruiser can be practically applied to large data centers and server farms.

8. RELATED WORK

Countermeasures against buffer overflow attacks: Over the past few decades, there has been extensive research in this area. In our previous work Cruiser [59], we divided existing countermeasures against buffer overflow attacks into seven categories: (1) buffer bounds checking [57, 18, 4, 25, 36, 44, 2, 15, 55, 5], (2) canary checking [12, 24, 42], (3) return address shadow stack or stack split [51, 10, 40, 20, 58], (4) non-executable memory [54, 49], (5) non-accessible memory [22, 56, 19], (6) randomization and obfuscation [7, 54, 11, 6], and (7) execution monitoring [29, 1, 9, 13, 45]. We refer the readers to Cruiser [59] for more details. Few countermeasures are suitable for high performance kernel heap buffer overflow monitoring and no one has been deployed in production systems.

Our work falls under the category of canary checking. Canary was firstly proposed in StackGuard [12], which tackles stack-smashing attacks by putting a canary word before the return address on stack. A buffer overflow that overwrites the return address would corrupt the canary value first. The approach has been integrated into GCC and Visual Studio. Robertson et al. [42] applied canary to protecting heap buffers. A canary is placed in the beginning of each heap chunk. When a heap buffer is overrun, the canary of the adjacent chunk is corrupted, which, however, is not detected until the adjacent chunk is coalesced, allocated, or deallocated; i.e., the detection relies on the control flow. Linux kernel provides a similar debugging option [3] and has the limitation likewise. Our approach enforces a constant concurrent canary checking and thus does not have the limitation. In addition, combining the stealth technique protecting the canary generation keys, the Signed Canary proposed in this paper can resist the attacks inferring canaries based on leaked ones.

Our previous work Cruiser [59], among the existing countermeasures, first proposed concurrent buffer overflow cruising in user space using custom lock-free data structures and non-blocking algorithms; the average performance overhead on SPEC CPU2006 is 5%. Two variants of Cruiser were implemented, namely Eager Cruiser and Lazy Cruiser. The former may incur false positives; although the probability ($1/2^{64}$ for 64-bit OSes) is extremely low it may be inappropriate in kernel space. The latter delays the deallocation of heap buffers, which is not desirable considering the narrow kernel memory address space. Unlike Cruiser that hooks per heap buffer allocation and deallocation, Kruiser interposes the much less frequent operations that switch pages into and out of the heap page pool and thus further reduces overhead. The settings and monitoring algorithms are also very different.

Virtual-machine introspection: Garfinkel and Rosenblum [21] first proposed the idea of performing intrusion detection from outside the monitored system. Since then, out-of-VM introspection has been applied to control-flow integrity checking [39, 46], malware prevention, detection, and analysis [30, 27, 16, 38, 31, 8, 41, 32, 23, 17], and attack replaying [28]. They monitor static memory areas (e.g. kernel code, Interrupt Description Table), interpose specific events

such as page faults, trace system behaviors, or detect violations of invariants between data structures. Considering the volatile properties of heap buffers, these approaches are infeasible or impractical for heap buffer overflow detection; for example, it is not practical to interpose every memory write on heap. Some approaches detected buffer overflow attacks as a side effect by detecting corrupted pointers or control flows, but cannot deal with non-pointer and non-control data manipulation on heap buffer objects. Approaches, such as kernel memory mapping and analysis, can be misled by buffer overflow attacks or perform better without heap corruption. Our approach can be complementary to them providing lightweight heap buffer overflow detection.

In contrast to out-of-VM monitoring, SIM [48] puts the monitor back into the VM and enables secure in-VM monitoring by providing discriminative memory views for the monitored system and the monitor. The stealth technique proposed in this work can be adapted to protect the keys, data structures and interposition code.

The most relevant work to our approaches is OSck [23]. OSck conducts partially out-of-VM concurrent monitoring to verify type-safety of data structures. It accesses existing heap metadata without acquiring locks and handles race conditions by suspending the monitored VM for double-check to avoid false positives. Our monitor runs concurrently with the monitored VM without incurring false positives and thus does not need to suspend the system for recheck.

9. CONCLUSION

We have presented KRUISER, a semi-synchronized concurrent kernel heap monitor that cruises over heap buffers to detect overflows in a non-blocking and out-of-VM manner. Unlike traditional techniques that monitor volatile memory regions with security enforcement inlined into normal functionalities (interposition) or by analyzing memory snapshots, we perform constant monitoring in parallel with the monitored VM on its live memory without incurring false positives. Our evaluation has shown that Kruiser is practical: it imposes negligible performance overhead on the system running SPEC CPU2006 and 2.7% throughput reduction on Apache. The concurrent *kernel cruising* approach leverages increasingly popular multicore architectures; its efficiency and scalability show that it can be applied to data centers and server farms in practice.

10. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05*, pages 340–353.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Usenix Security '09*, pages 51–66.

[3] P. Argyroudis and D. Glynos. Protecting the core: Kernel exploitation mitigations. In *Black Hat Europe '11*.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI '04*, pages 290–301.

[5] K. Avijit and P. Gupta. Tied, libsafeplus, tools for runtime buffer overflow protection. In *Usenix Security '04*, pages 4–4.

[6] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03*, pages 281–289.

[7] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Usenix Security '03*, pages 105–120.

[8] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. CCS '09, pages 555–565.

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI '06*, pages 147–160.

[10] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS '01*, pages 409–417.

[11] C. Cowan and S. Beattie. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Usenix Security '03*, pages 91–104.

[12] C. Cowan and C. Pu. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security '98*, pages 63–78, January 1998.

[13] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *Usenix Security '06*, pages 105–120.

[14] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace & kernelspace. In *Usenix Security '08*, pages 395–410.

[15] E. D.Berger. HeapShield: Library-based heap overflow protection for free. Tech. report, Univ. of Mass. Amherst, 2006.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. CCS '08, pages 51–62.

[17] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. Oakland '11.

[18] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI '03*, pages 155–167, June 2003.

[19] E. Fence. Malloc debugger. http://directory.fsf.org/project/ElectricFence/.

[20] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Usenix Security '01*, pages 55–66.

[21] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS '03*, pages 191–206.

[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *the Winter 1992 Usenix Conference*, pages 125–136.

[23] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. ASPLOS '11, pages 279–290.

[24] IBM. ProPolice detector. http://www.trl.ibm.com/projects/security/ssp/.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Usenix ATC '02*, pages 275–288, June 2002.

[26] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *the International Workshop on Automatic Debugging*, 1997.

[27] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. Usenix ATC '06.

[28] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. SOSP '05, pages 91–104.

[29] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Usenix Security '02*, pages 191–206.

[30] K. Kourai and S. Chiba. HyperSpector: virtual distributed monitoring environments for secure intrusion detection. VEE '05, pages 197–207.

[31] A. Lanzi, M. I. Sharif, and W. Lee. K-Tracer: A system for extracting kernel malware behavior. In *NDSS '09*.

[32] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. NDSS '11.

[33] T. Mandt. Kernel pool exploitation on Windows 7, 2011. https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf.

[34] P. E. Mckenney. Memory barriers: a hardware view for software hackers, 2009.

[35] D. Mosberger. Memory consistency models. *Operating Systems Review*, 17(1):18–26, January 1993.

[36] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[37] NIST. National Vulnerability Database. http://nvd.nist.gov/.

[38] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. Oakland '08, pages 233–247.

[39] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. CCS '07, pages 103–115.

[40] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Usenix ATC '03*, pages 211–224.

[41] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. RAID'10, pages 178–197.

[42] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *LISA '03*, pages 51–60.

[43] D. Roethlisberge. Omnikey Cardman 4040 Linux driver buffer overflow, 2007. http://www.securiteam.com/unixfocus/5CP0D0AKUA.html.

[44] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS '04*, pages 159–169.

[45] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys '09*, pages 33–46.

[46] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. SOSP '07, pages 335–350.

[47] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

[48] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. CCS '09, pages 477–487.

[49] Solar Designer. Non-executable user stack, 1997. http://www.open wall.com/linux/.

[50] sqrkkyu and twzi. Attacking the core: Kernel exploiting notes, 2007. http://phrack.org/issues.html.

[51] StackShield, 2000. http://www.angelfire.com/sk/stackshield/.

[52] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *EUROSEC '09*, pages 1–8.

[53] C. S. Technologies. OpenBSD IPv6 mbuf remote kernel buffer overflow, 2007. http://www.securityfocus.com/archive/1/462728/30/0/threaded.

[54] The PaX project. http://pax.grsecurity.net/.

[55] T. K. Tsai and N. Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *DSN '02*, pages 541–541.

[56] Valgrind. http://valgrind.org/.

[57] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS'00*, pages 3–17.

[58] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks. In *Workshop Evaluating & Architecting Sys. Depend.*, 2002.

[59] Q. Zeng, D. Wu, and P. Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *PLDI '11*. To appear.