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

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

[†]School of Computer Science, Beijing Institute of Technology, Beijing, China
{dhai, chzhoo}@bit.edu.cn
[*]Pennsylvania State University, University Park, PA, USA
{quz105, dinghao, pliu}@psu.edu

Abstract

Kernel heap buffer overflow vulnerabilities have been exposed for decades, but there are few practical countermeasure that can be applied to OS kernels. Previous solutions either suffer from high performance overhead or compatibility problems with mainstream kernels and hardware. In this paper, we present KRUISER*, a concurrent kernel heap buffer overflow monitor. Unlike conventional methods, the security enforcement of which is usually inlined into the kernel execution, Kruiser migrates security enforcement from the kernel's normal execution to a concurrent monitor process, leveraging the increasingly popular multi-core 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 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 process out of the monitored VM, while heap memory allocation information is collected inside the monitored VM in a secure and efficient way. The hybrid VM monitoring technique combined with the secure canary that cannot be counterfeited by attackers provides guaranteed overflow detection with high efficiency. 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.*

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 [39].

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 kernel-level protection mechanism. Recently, more and more realistic buffer overflow exploits have been released in modern operating systems including Linux [52], OpenBSD [54] and the latest Windows 7 system [35].

Many effective countermeasures against stack-based buffer overflows have been proposed, some of which, such as StackGuard [14] and ProPolice [26], have been widely deployed in compilers and commodity OSes. On the other hand, practical countermeasures against heap-based buffer overflows are few, especially in the 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, 27, 38, 28, 47], 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 [45, 3], so that the detection occasions entirely depend on the control flow, which may allow a large time window for attackers to compromise the system. Other approaches [48, 16] 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 monitor. 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. The monitor process is constantly checking kernel heap buffer overflows in an infinite loop. We call this technique *kernel cruising*. 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 multi-core architecture.

We have explored kernel heap management design properties to collect heap buffer region information at page level instead of individual buffers. A conventional 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 using a dynamic data structure, our approach avoids 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.

To provide performance isolation and prevent the monitor process from being compromised by attackers, we take advantage of virtualization to deploy the monitor process in a trusted execution environment. Kruiser employs the Direct Memory Mapping technique, by which the monitor process can perform frequent memory introspection efficiently. On the other hand, the buffer address information is collected inside the VM to avoid costly hypervisor calls; Secure In-VM (SIM) [50] approach is adapted to protect the metadata from attackers.

In summary, we make the following contributions:

- **Semi-synchronized concurrent monitoring:** We propose a novel non-blocking concurrent monitoring algorithm, in which neither the monitor process nor the monitored process needs to be fully synchronized to eliminate concurrency issues such as race conditions; the monitor keeps checking live kernel memory without incurring false positives. We call this *semi-synchronized*. Concurrent monitoring leverages more and more popular multicore architectures and thus the performance overhead is low compared to inline security enforement.

- **Kernel cruising:** The novel cruising idea has been recently explored [65, 25]. 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 metadata 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-VM monitoring plus In-VM interposition:** The isolated monitor process along with direct memory mapping through virtualization is applied to achieve efficient out-of-the-box monitoring. Moreover, we apply the SIM framework to protect the In-VM metadata collection. The hybrid VM monitoring scheme provides a secure and efficient monitoring.

- **Secure canary:** Unlike conventional canaries, which can be inferred and then counterfeited based on other canary values, we proposed the conception of *secure canary* and provided an efficient solution, such that once a canary is corrupted, it cannot be recovered by attackers. Secure canaries along with the hybrid VM monitoring scheme guarantee the detection of buffer overflow attacks.

We have implemented a prototype of Kruiser based on Linux and the Xen hypervisor. The effectiveness of Kruiser has been evaluated and the experiments show that Kruiser can detect kernel heap overflows effectively. With respect to performance and scalability, our kernel cruising approach is practical—it imposes negligible performance overhead on SPEC CPU2006, and the throughput slowdown on Apache is 7.9% on average.

2 Threat Model

This paper is focused on monitoring kernel heap buffer overflows. Other security issues, such as memory content disclosure or shellcode injection by exploiting format string vulnerabilities, are not in the scope of this paper. We assume the goal of an attacker X is to compromise the kernel of a VM; then he can do anything the kernel can do. Attacker X can launch arbitrary attacks against the kernel, but we assume that before a heap overflow attack succeeds, the kernel has not been compromised by other attacks launched by X. Otherwise, he had already achieved his goal. Once the attacker X has compromised the kernel using heap buffer overflows, we assume X can do anything the kernel is authorized to do regarding memory read/write, OS control flow altering, etc. Since this work relies on the virtualization technology to monitor the kernel heap, we assume the underlying hypervisor is trusted. We leverage previous research (e.g., HyperSafe [62], HyperSentry [6] and HyperCheck [61]) to protect the hypervisor security. Moreover, our trusted computing base includes a trusted VM where the monitor resides. This VM is special, and it is not supposed to run any other applications.

3 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 usually 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. Nevertheless, 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 [25], 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 contention between accessing processes may still lead to high overhead.

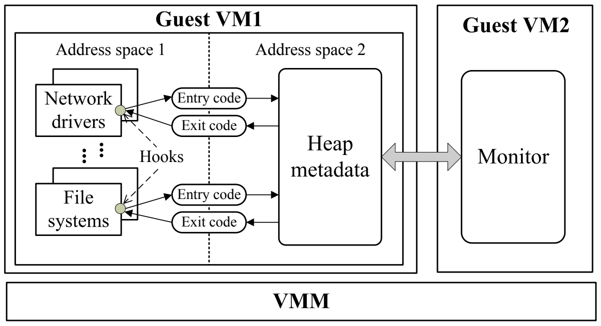


Figure 1. Overview of Kruiser.

C2. Self-protection and canary counterfeit. 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. After the system is compromised by exploiting the buffer overflow vulnerabilities, attackers may try to kill the monitor process to disable the detection completely. Attackers can also tamper or manipulate the data structure needed by our component to mislead or evade the detection. Moreover, attackers may try to recover the canary after corrupting it.

C3. Compatibility. Kernel heap management is among 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 mainstream systems as well as hardware.

4 Overview

Kruiser attaches one canary word at the end of each heap buffer and generates 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 different VM from the monitored OS to strengthen self-protection. The heap buffer metadata and hooking code are kept in the monitored VM to achieve efficient buffer information collection. The monitor accesses the inter-VM heap metadata via an efficient technique called direct memory mapping. To achieve a concurrent monitoring, the monitor process needs to locate and access the canaries reliably and efficiently, while the monitored system allocates and deallocates the buffers and heap pages continuously.

To address the synchronization challenge (C1), We explore the characteristics of kernel heap management, and propose to interpose heap page allocation and deallocation, by which we maintain concise metadata describing canary locations in a separate efficient data structure. Compared with interposing 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 with 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 and canary counterfeit 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 the heap metadata and interposition code from being compromised by attackers, we apply the SIM [50] framework, which enables the data and code to be protected safely and efficiently inside the monitored VM. As shown in Figure 1, by utilizing the VMM, we introduce two separate address spaces in the monitored VM, and address space 2 is used to place the heap metadata and interposition code. The entry code and exit code are the only ways to transfer execution between the two address spaces so that the metadata can be updated. Canaries are generated applying efficient cryptography, such that once a canary is corrupted, it is difficult for attackers to infer and then recover the canary value.

To address the compatibility challenges (C3), we made minimal changes to the existing kernel heap management based 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, so that kernel heap buffer allocation algorithms are not changed. On the other hand, the major monitor component is located out of the monitored kernel leveraging the popular VMM platform, which is widely used in cloud computing nowadays.

5 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 5.1. We discuss potential race conditions in Section 5.2 and describe our algorithm in Section 5.3.

5.1 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 [25], 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 in our case.

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 predetermined. 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 6.

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. 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)[1] 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. The traverse along pages is suitable for 32-bit OSes with small kernel memory address space.

For 64-bit systems with large address space and physical memory, the flat PIA structure may not be scalable enough, and sparse kernel heap pages could lead to a concern of significant ineffective scanning. We will present an extended form of PIA structure in Section 8.1, which could solve the sparse heap pages problem with high scalability and low memory overhead.

5.2 Race conditions

Exploring the characteristics of kernel heap management, we proposed the static PIA structure, which avoids heap monitoring from relying on kernel-specific heap data structure 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 heap cruising algorithm, we first discuss the potential race conditions for sharing the PIA structure, which motivate our semi-synchronized design in Section 5.3. 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 updating a PIA entry is not atomic, a race condition occurs if we allow multiple processes to modify 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 values.

Time of check to time of use (TOCTTOU): For a given 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.

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

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 updated since last access 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 intends to determine whether the entry has been updated by reading the flag twice, considering that other processes may have removed the page from the heap page pool and then added 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 based on PIA to resolve all the problems without incurring false positives or high overhead.

5.3 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

```
 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.

has its own synchronization mechanisms to prevent one page frame from being manipulated for inconsistent purposes at the same time. For example, Linux functions 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 updating 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 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 [37] 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 [36], 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*. 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.

6 System Design and Implementation

6.1 Background

Linux adopts the *slab* allocator[2] for kernel heap management. It uses *caches* to organize heap buffer objects. There are two types of caches in kernel heap, namely *general caches* and *specific caches*. General caches are mainly used to serve kmalloc calls requesting heap buffers of various sizes, while each specific cache is used to allocate objects of a specific kernel data structure, such as task_struct. A cache consists of one or more *slabs*, each of which occupies one or more physically contiguous pages and contains

[2]Similar schemes are widely used in other commodity systems, such as Solaris and FreeBSD.

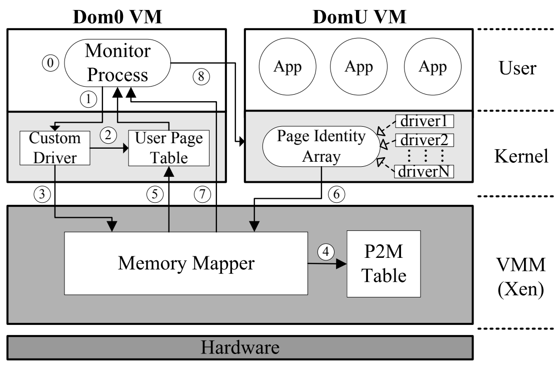


Figure 3. Kruiser Architecture. The numbers in the small circle indicate Kruiser's work flow.

objects of the same type. When a slab is created to serve a buffer request, additional objects are created in the slab's memory pages to serve further buffer requests.

6.2 Architecture

The architecture, as shown in Figure 3, can be divided into three parts: VMM, Dom0 VM, and DomU VM. The Monitor Process in Dom0 VM executing Monitor (Figure 2) in an infinite loop to monitor the kernel of DomU VM. A tiny component, namely Memory Mapper, inside the VMM is used to map the kernel memory of the monitored VM to the monitor process, which is detailed in Section 6.3. The custom driver in Dom0 VM is used to assist the monitor process to release extra memory during the memory mapping. The Page Identity Array and the interposition code inside AddPage and RemovePage (Figure 2) reside in the kernel space of DomU VM, whose protection is presented in Section 6.4.

The out-of-VM monitoring ensures performance isolation and secureness, but usually leads to high overhead. The in-VM information collection provides native code execution and memory access environments, but may be vulnerable to attacks. By addressing the problems, we combine the two schemes as a hybrid solution to provide a secure and efficient monitoring.

6.3 Direct Memory Mapping

To achieve an out-of-the-box monitoring, a conventional method is to run a monitor process in a trusted VM 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 monitored VM's page table and map the target machine frames to be accessible from the

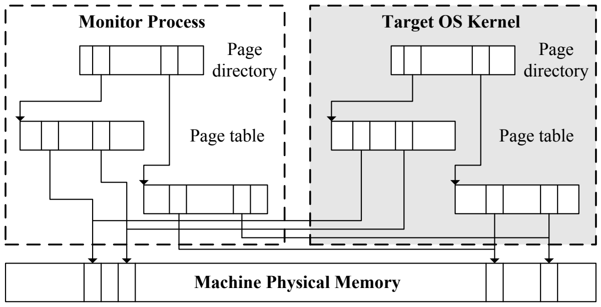


Figure 4. The direct memory mapping mechanism.

monitor process. To avoid 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 that the VMM manipulates the page table of the monitor process such that the monitor process can access the kernel memory of the monitored OS directly, as illustrated in Figure 4. Note that the custom driver is implemented as a loadable kernel module, such that the Dom0's kernel code is not modified. The procedure of DMM can be divided into three stages.

First, the Monitor Process allocates a chunk of memory whose size is determined by the maximum number of memory pages used for DomU VM's kernel heap (⓪ in Figure 3). As Linux kernel heap only resides in physically contiguous memory areas, its maximum size is less than 896MB in 32-bit kernels even if the physical memory size is larger than 896MB. The goal of this stage is to create a contiguous range of virtual addresses. By properly manipulating the page table entries (PTEs), the VMM enables the monitor process to access the memory of the target OS kernel within the monitor's virtual address space. However, due to the demand paging mechanism, actually the memory for PTEs are not allocated when the virtual addresses are created. Therefore, we need to access the created memory chunk to trigger the creation of PTEs before operating on them.

Second, the Monitor Process notifies the Custom Driver to reclaim the newly allocated pages (①) with the PTEs retained. This is necessary because the Monitor Process only needs the new virtual addresses and the corresponding PTEs but does not use the allocated pages; returning these pages back can save a lot of memory. Specifically, this stage consists of four steps. 1) The Custom Driver first walks the page table of the Monitor Process to identify the PTEs for the memory chunk allocated in the first stage (②). 2) Then, with these identified PTEs, the Custom Driver searches for the corresponding page descriptors used by the page frame management. 3) After that, the Custom Driver clears the relevant flags in 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 reclaiming 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 collects all the MFNs of the DomU. With the mapping information, the Memory Mapper updates the PTEs of the Monitor Process accordingly. Specifically, given the newly allocated virtual address range, the Memory Mapper walks the User Page Table to find the corresponding PTEs (⑤), whose page frame numbers are then changed to the MFNs that are collected from the P2M table. In this way, the Monitor Process can access the entire kernel of the target OS with its own page table.

Once the Page Identity Array is allocated and initialized in DomU VM, 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).

Reducing TLB Pressure. As the memory area that the Monitor Process accesses may be large when a lot of kernel slabs are produced, the kernel cruising may incur high TLB pressure. 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). Note that to this end 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 to work 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.

6.4 In-VM Protection

Since the PIA data structure (metadata) and the interposition code reside in the kernel space of DomU VM, attackers may manipulate them directly after exploiting buffer overflow vulnerabilities. To solve this problem, a conventional method is to move the data structure and code to be protected into the hypervisor or another trusted VM. However, it will incur significant performance overhead when the world switches between the hypervisor and the VM become frequent, especially for such fine-grained monitoring as in our case. Instead, we employ the SIM [50] framework, which enables a secure and efficient in-VM monitor-

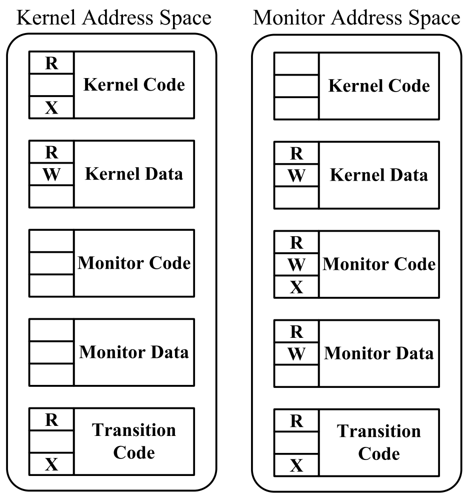
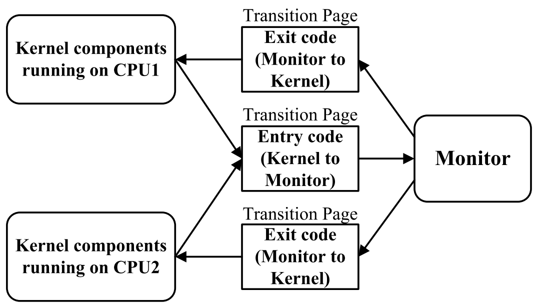


Figure 5. Memory protections in the kernel and monitor address space.



Kernel components running on CPU1
Transition Page
Exit code (Monitor to Kernel)
Transition Page
Entry code (Kernel to Monitor)
Monitor
Kernel components running on CPU2
Transition Page
Exit code (Monitor to Kernel)

Figure 6. Address space switching via transition pages in SMP.

ing. Specifically, the hypervisor creates a separate protected address space inside DomU VM and puts the code and data to be protected in it, such that those memory regions are protected from the DomU VM kernel by the hypervisor, and the separate address space can only be entered and exited through specially constructed protected gates.

In our case, we need to move the interposition code added in the critical section of AddPage and RemovePage as well as the PIA data structure in Figure 2 to the protected memory regions. To this end, we construct two shadow page tables (SPTs) specifying different access permissions for the kernel and the In-VM monitor part.[3] As shown in Figure 5, within a kernel address space, a process is not allowed to access the monitor code and data regions, while the kernel code cannot be executed after a process switches to the monitor address space. To invoke the monitor's code in the kernel address space, the transition code is used to switch address spaces and is executable in both address spaces. The transition code modifies the *CR3* register, which contains the physical address of the root of the target shadow page table. By default, any change of *CR3* will result in a *VMExit*. Fortunately, a recent hardware feature allows us to change the *CR3* without being trapped to the hypervisor if its value is in the *CR3_TARGET_LIST*, which is maintained by the hypervisor.

Address Space Maintaining and Switching in SMP. Maintaining and switching shadow page tables in Symmetric Multi-Processing (SMP) involves two challenges: 1) The SPTs for the kernel address space and the monitor address space should get synchronized for correctness. 2) The transition code should determine the correct *CR3* target when switching back to the kernel address space.

To address the first challenge, our approach explores the observation that the monitor only needs to access the kernel heap (for placing canaries) and the kernel stack (for accessing the arguments and storing local variables), which only reside in non-paged contiguous memory areas. Hence, by looking up the P2M table, we can build the memory mapping in the SPT used by the monitor with one-time effort, and then no synchronization is needed.

As to the second challenge, although one common transition page for the entry code is sufficient, one transition page for the exit code is needed for per processor, considering that different processor may have entered the monitor address space from different process address spaces. In each transition page, the *CR3* address to be assigned has to be equal to the address of the shadow page directory that was used by the current processor prior to entering the monitor address space, as shown in Figure 6. To this end, we modify hypervisor to update the *CR3* target used in the associated exit code when a processor performs process switches. The hypervisor should also update the *CR3_TARGET_LIST* accordingly. To facilitate the monitor to select the corresponding transition page when switching back, we generate these pages according to the different *CPU ID*, which can be easily determined by the monitor (i.e., using the function smp_processor_id()).

Security Check. By invoking duplicate AddPage for the corrupted page, attackers can recover the canaries. To avoid this problem, we add one more check in the protected code to prevent pages with odd PIA version numbers from being added again. On the other hand, if attackers invoke RemovePage maliciously, the final round of canary checking in the function can detect overflows.

Additionally, we need to consider an attack scenario where the exploit installs something (e.g., rootkit) on the system and then reboots the kernel so that it would by-

[3]Note that the In-VM monitor part only includes the PIA and the interposition code and will be referred to as the *monitor* in this section for short, while the monitor process still runs out of the VM.

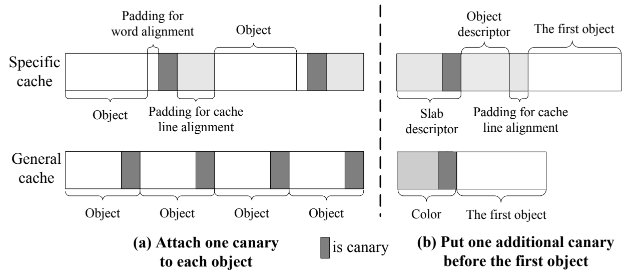

Padding for word alignment
Object
Specific cache
Object
Padding for cache line alignment
General cache
Object
Object
Object
Object
(a) Attach one canary to each object
is canary
Object descriptor
The first object
Slab descriptor
Padding for cache line alignment
Color
The first object
(b) Put one additional canary before the first object

Figure 7. Placing canaries into kernel objects.

pass our detection mechanism. To address this problem, we could utilize the hypervisor to mediate the reboot. Before the kernel is rebooted, we can pause the system for a while such that the monitor process will discover the corrupted canaries after scanning the entire kernel heap.

6.5 Placing Canaries

To detect underflows as well as overflows, it is straightforward to place two canaries surrounding each buffer. Actually, Linux with slab debug-enabled version has adopted this scheme to place canaries. Unfortunately, this method does not make kernel objects aligned in the first-level hardware cache, which may result in more cache misses. To overcome this limitation, we only use one canary instead of two canaries to surveil each kernel object. Since the same type of kernel objects are grouped together inside a slab, our approach can still detect the heap underflow attack occurred in one object (but not the first one in a slab) by checking the canary attached by the previous object.

As shown in Figure 7(a), we apply two different ways to place the canary. For the specific caches, we first pad the objects to be word-aligned in size. Then, we add one word canary following the object. Finally, to ensure the object get L1 cache line aligned, we put some additional padding at the end of this object. On the other hand, as the objects in general caches have already got L1 cache line aligned in size, there is no need to change the form of these objects. Instead, we place a canary in the last word of each object. In addition, we hook the general object allocation function (i.e., kmalloc), and increase the original requested size by one word to hold the canary.

Although the scheme above works well to detect underflows (and overflows), it cannot deal with underflows occurred in the first object, as there is no canary preceding it. To tackle this issue, as shown in Figure 7(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 is placed at the end of this slab descriptor; or if there is a slab color,[4] we put a canary in the last word of this color.

Secure Canary Generation. To set canary values for kernel objects, a practical solution should meet the two requirements: R1) after attackers have compromised the monitored kernel via buffer overflows, they cannot recover the corrupted canaries; R2) The canary generation and verification algorithms should be efficient so that they will not affect the system performance and detection latency. To satisfy these requirements, we employ a stream cipher (RC4 [63]) to generate canary values. For each slab, we first extract a random number from the entropy pool in Linux. Then, this random number is used as the key “stretched” by RC4 into a stream of bytes, the length of which is decided by the number of objects inside the slab. Finally, each 4 bytes of this stream is selected as a canary value for each object. On the other hand, regarding canary checking, we store the key (i.e., the random number) into the corresponding PIA entry for each slab.

Guaranteed Detection. With the In-VM protection and secure canary generation, attackers can not hide their attacks in that 1) The In-VM protection prevent attackers from manipulating the PIA entries; 2) The canary generation based on the stream cipher guarantees the difficulty for attackers to recover the corrupted canaries within one cruising cycle. In addition, attackers cannot change the memory mapping between the monitor process and the monitored kernel in that the associated page table is maintained by another trusted VM (i.e., Dom0). Therefore, the attacks are bound to be detected within one cruising cycle after compromising the system, unless the attackers know the exact canary value to be corrupted beforehand, which usually implies the overread and overrun vulnerabilities overlap for exactly the same buffer area and which is very rare.

6.6 Locating Canaries

To locate and verify canaries in the Monitor Process, we hook the slab allocations and deallocations to store the metadata into the PIA entries, one of which is shown in Figure 8. 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 start address of one object and its actual object size, we can get the canary address. To acquire the start 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

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

```
struct PIA_entry{
    unsigned int version;
    short mem; // the starting address of the first object
    short slab_size; // the size of the slab descriptor
    int obj_size; // the actual size used by each object
    int buffer_size; // the whole size for each object
    int number; // the number of objects in this slab
    long key; // the key for canary verification
};
```

Figure 8. PIA entry.

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 predetermined. 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.

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 24 bytes in our implementation, the memory overhead is as low as 24/4096. Furthermore, it is possible to reduce the size of the PIA entry by packing its fields.

7 Evaluation

To evaluate Kruiser, we developed a prototype of Kruiser based on 32-bit Linux and the Xen hypervisor (with PAE enabled), and conducted effectiveness tests and measured performance overhead. All the experiments were run 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.

7.1 Effectiveness

To test whether Kruiser can detect heap buffer overflows, we deliberately introduced three explicit vulnerabilities [46, 52] 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, such that malicious users 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 [52]. 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.

In addition to the synthetic attacks, we also exploited two real-world heap buffer overflow vulnerabilities [57, 58] in Linux. For the first one, we sent particularly crafted ASN.1 BER data to trigger a heap overflow. In the second test, we used a special eCryptfs file whose encrypted key size is larger than ECRYPTFS_MAX_ENCRYPTED_KEY_BYTES to overflow a buffer. Kruiser detected all the realistic overflows.

The above experimental results indicate that Kruiser is effective in defending against kernel heap buffer overflow attacks.

7.2 Performance and Scalability

To evaluate the performance of our monitoring mechanism, we carried out a set of experiments. Each of these experiments was conducted in three different environments, including original Linux, Kruiser with SIM protection (referred as SIM-Kruiser subsequently), and Kruiser without SIM protection.

In the first experiment, we executed the SPEC CPU2006 Integer benchmark suite. Figure 9 shows that the average performance overhead for both Kruiser and SIM-Kruiser are negligible. When the slab allocation is frequent, the

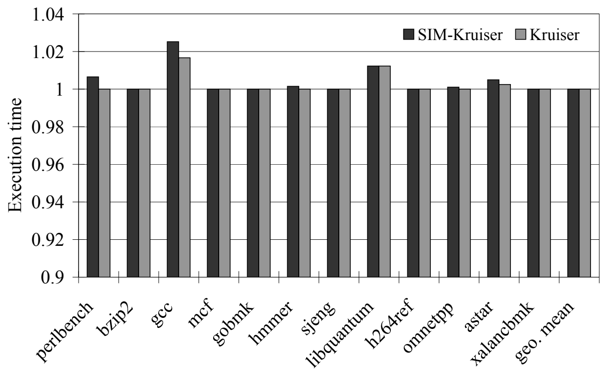


Figure 9. SPEC CPU2006 performance (normalized to the execution time of original Linux).

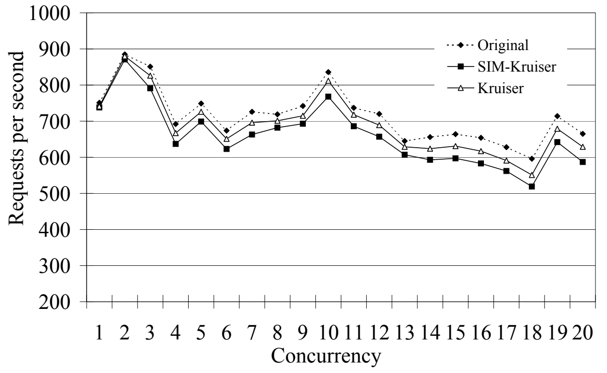


Figure 10. Throughput of the Apache web server for varying numbers of concurrent requests.

performance overhead is a little bit higher, such as in gcc; however, the maximal performance overhead is less than 3%.

For the scalability measurement, we tested the throughput of the Apache web server with concurrent requests. Specifically, we ran Apache 2.2.8 to serve a 3.7KB html web page. We used ApacheBench 2.3 running 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, and we observed that the number of the kernel heap buffer object allocation increases along with the concurrency level. As shown in Figure 10, the performance overhead imposed by Kruiser and SIM-Kruiser are both relatively stable. On average, Kruiser only incurs about 3.8% performance degradation and SIM-Kruiser about 7.9%.

Table 1. 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	107,824	105,145	106,378	39,259
bzip2	79,085	76,325	76,682	27,662
gcc	78,460	76,810	77,413	27,774
mcf	82,885	79,328	79,540	28,156
gobmk	80,761	80,345	80,519	28,606
hmmer	81,278	80,435	80,591	28,635
sjeng	81,437	80,259	80,535	28,610
libquantum	80,911	80,317	80,407	28,493
h264ref	80,756	80,337	80,480	28,572
omnetpp	82,109	80,796	81,088	28,836
astar	81,592	81,022	81,097	28,897
xalancbmk	99,436	82,747	88,454	30,190

7.3 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, in order to evaluate the detection latency, which is less than or equal to the cruising cycle at the attack time. As shown in Table 1, 10 of 12 applications' average cruising cycles are shorter than 29 ms, and the other two applications' are below 40 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 objects in general caches. This is practical because attackers mainly exploit this category of buffers in the real world.

8 Discussion

8.1 Scalable Monitoring

For 32-bit OSes the flat PIA array structure is feasible. However, for a 64-bit system with TBes of physical memory, the memory overhead due to the PIA structure may be not desirable. In addition, when page frames serving for the kernel heap are sparse, Kruiser has to walk a long distance before encountering a heap page, which implies heavy ineffective checking.

Both concerns can be resolved by extending the PIA array to a multi-level table structure, the idea of which is inspired by the page table structure. The first-level PIA table is a simple array occupying one page frame. Each entry of a PIA table except for the last-level one stores the address of a next-level table and other information including the count of non-zero entries in the next-level table; the structure of last-level PIA tables are the same as a PIA array. If the count is zero, the entry does not point to any PIA table.

Specifically, when a page with address A is to be added into the heap page pool, the most significant serveral bits of A are used as an index to locate the entry in the first-level PIA table. If the entry is empty, a next-level PIA table is allocated and the entry is filled with the address of the new PIA table and the count value 1. The remaining bits of A are used to locate the following levels of PIA table, until the entry in the last-level PIA table is located, and the metadata of page A is then recorded there. For a 64-bit system, a three-level PIA structure suffices. When the monitor process traverses along the PIA directory using a depth-first-search algorithm, it bypasses empty PIA entries corresponding large bulks of contiguous pages. To prevent race conditions when multiple processes accessing the same PIA entry, CAS (Compare-And-Swap) instructions are needed.

The extended multi-level PIA structure not only reduces memory overhead but also accelerate the cruise cycle. It is similar to the multi-level page table structure; like the page table used in 64-bit systems, the PIA structure and accordingly the monitoring are scalable for systems with large address spaces and physical memory.

8.2 Viable Deployment

Large data centers using shipping-containers packed with thousands of servers each are common 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. Moreover, one Kruiser instance is able to monitor multiple VMs given an acceptable detection latency much longer than the cruising cycle, without affecting the guaranteed detection property. In addition, the performance isolation provided by the underlying VMM ensures the monitor process and the monitored VM do not abuse computing resources to interfere with each other, which is a desirable property for users.

With the popularity of multi-core architectures, servers built with many cores are more and more common. The hardware evolution trend embraces the concurrent monitoring fashion, as the cost for a unit core running a monitor instance decreases sharply, and the extra energy consumption by one core is relatively low for machines with hundreds of cores. Therefore, the scalability and low cost properties imply that Kruiser can be practically applied to large data centers and server farms.

9 Related work

9.1 Countermeasures Against Buffer Overflows

Over the past few decades, there has been extensive research in this area. We divided existing countermeasures against buffer overflows into seven categories: (1) buffer bounds checking [60, 20, 4, 27, 38, 47, 2, 17, 56, 5], (2) canary checking [14, 26, 45], (3) return address shadow stack or stack split [53, 12, 43, 22, 64], (4) non-executable memory [55, 51], (5) non-accessible memory [24, 59, 21], (6) randomization and obfuscation [9, 55, 13, 7], and (7) execution monitoring [31, 1, 11, 15, 48]. Few countermeasures are suitable for high performance kernel heap buffer overflow monitoring and no one has been deployed in production systems.

Kruiser falls into the category of canary checking. Canary was firstly proposed in StackGuard [14], 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. [45] applied canary to protecting heap buffers. 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. Our approach enforces a constant concurrent canary checking and thus does not have the limitation. In addition, the *secure canary* conception is innovative.

The previous work Cruiser [65], among the existing countermeasures, first proposed concurrent buffer overflow cruising in user space using custom lock-free data structures. Unlike Cruiser that hooks per heap buffer allocation and deallocation, Kruiser explores the characteristics of kernel heap management to interpose the much less frequent operations that switch pages into and out of the heap page pool, such that our system relies on on a fix-sized array data structure instead of the lock-free data structures to maintain the metadata. The monitoring algorithms are thus very different. In addition, the hybrid monitoring scheme differs a lot from the user space monitoring.

Compared with the methods based on probabilistic memory safety (e.g., DieHard [8] and DieHarder [40]), Kruiser imposes negligible performance overhead. Nevertheless, Kruiser focuses on kernel heap, while DieHard and DieHarder have only been demonstrated for user-space programs. Our previous work Cruiser [65] on user-space buffer overflow monitoring presents detailed comparison with DieHarder on performance for the SPEC CPU2006 benchmark. In addition, DieHard and DieHarder consume more memory than Kruiser, which may be a problem for kernel.

9.2 Virtual Machine Introspection

Garfinkel and Rosenblum [23] first proposed the idea of performing intrusion detection from outside of the monitored system. Since then, out-of-VM introspection has been applied to control-flow integrity checking [42, 49], malware prevention, detection, and analysis [32, 29, 18, 41, 33, 10, 44, 34, 25, 19], and attack replaying [30]. 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 for kernel heap buffer overflow monitoring; for example, it is impractical to interpose every memory write on the 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 [50] 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. Our approach makes use of this technique to protect the heap metadata, while the monitor process still runs out-of-VM to achieve parallel monitoring, leveraging the multiprocessor architecture. The hybrid scheme enables a secure and efficient monitoring.

OSck [25] also performs kernel space cruising for rootkit detection. As OSck does not synchronize the running kernel and the verification process, it needs to suspend the system when an anomaly is detected to avoid false positives, while our approach does not need to stop the world for detection. In addition, OSck does not check generic buffers allocated using `kmalloc`, which are common attack targets, while Kruiser checks the whole kernel heap.

10 Conclusion

We have presented KRUISER, a semi-synchronized concurrent kernel heap monitor that cruises over heap buffers to detect overflows in a non-blocking manner. Unlike traditional techniques that monitor volatile memory regions with security enforcement inlined into normal functionality (interposition) or by analyzing memory snapshots, we perform constant monitoring in parallel with the monitored VM on its live memory without incurring false positives. The hybrid VM monitoring scheme provides high efficiency without sacrificing the security guarantees. Attacks are bound to be detected within one cruising cycle. Our evaluation has shown that Kruiser imposes negligible performance overhead on the system running SPEC CPU2006 and 7.9% throughput reduction on Apache. The concurrent *kernel cruising* approach leverages increasingly popular multi-core architectures; its efficiency and scalability manifest that it can be deployed in practice.

Acknowledgement

We would like to thank Xi Xiong for his valuable comments during the system design and implementation, and also thank our shepherd Xuxian Jiang as well as the anonymous reviewers for their comments that helped shape the final version of this paper. This work was partially supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, and AFRL FA8750-08-C-0137. Donghai Tian was supported by the China Scholarship Council (No. 2009603047).

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] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[7] 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.

[8] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

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

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

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

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

[20] 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.

[21] Electric Fence. Malloc debugger. http://directory.fsf.org/project/ElectricFence/.

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

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

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

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

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

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

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

[34] 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.

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

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

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

[38] 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.

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

[40] G. Novark and E. D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

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

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

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

[44] 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.

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

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

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

[48] 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.

[49] 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.

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

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

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

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

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

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

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

[57] US-CERT/NIST. CVE-2008-1673. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1673.

[58] US-CERT/NIST. CVE-2009-2407. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2407.

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

[60] 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.

[61] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: a hardware-assisted integrity monitor. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[62] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[63] Wikipedia. RC4. http://en.wikipedia.org/wiki/RC4.

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

[65] Q. Zeng, D. Wu, and P. Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 367–377, New York, NY, USA, 2011. ACM.