Cruiser: Concurrent Heap Buffer Overflow Monitoring Using Lock-free Data Structures

Qiang Zeng
Department of Computer Science & Engineering, Pennsylvania State University, University Park, PA 16802
quz105@cse.psu.edu

Dinghao Wu
College of Information Sciences & Technology, Pennsylvania State University, University Park, PA 16802
dwu@ist.psu.edu

Peng Liu
College of Information Sciences & Technology, Pennsylvania State University, University Park, PA 16802
pliu@ist.psu.edu

Abstract

Security enforcement inlined into user threads often delays the protected programs; inlined resource reclamation may interrupt program execution and defer resource release. We propose *software cruising*, a novel technique that migrates security enforcement and resource reclamation from user threads to a concurrent monitor thread. The technique leverages the increasingly popular multicore and multiprocessor architectures and uses *lock-free* data structures to achieve *non-blocking* and efficient synchronization between the monitor and user threads. As a case study, software cruising is applied to the heap buffer overflow problem. Previous mitigation and detection techniques for this problem suffer from high performance overhead, legacy code compatibility, semantics loyalty, or tedious manual program transformation. We present a concurrent heap buffer overflow detector, CRUISER, in which a concurrent thread is added to the user program to monitor heap integrity, and custom lock-free data structures and algorithms are designed to achieve high efficiency and scalability. The experiments show that our approach is practical: it imposes an average of 5% performance overhead on SPEC CPU2006, and the throughput slowdown on Apache is negligible on average.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Security, Verification, Languages, Algorithms

Keywords Software cruising, buffer overflow, program monitor, multicore, concurrency, lock-free, non-blocking algorithms

1. Introduction

Despite extensive research over the past few decades, buffer overflow remains as one of the top software vulnerabilities. In 2009, 39% of the security vulnerabilities published by US-CERT [67] were related to buffer overflows. As of September 2010, 12 of the 20 most severe vulnerabilities ranked by US-CERT were buffer overflow related. Vulnerabilities listed by security websites such as SecurityFocus [60] and Securiteam [54] manifest a similar pattern. The related exploits, such as CodeRed [10] and SQLSlammer [12], have inflicted billions of dollars worth of damages [38].

A buffer overflow occurs when a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. There are mainly two types of buffer overflows according to the overflowed buffer's memory region, namely stack-based buffer overflows and heap-based buffer overflows.

Stack-based buffer overflows are the most exploited vulnerability, as the return addresses for function calls are stored together with buffers on stack. By overflowing a local buffer, the return address can be overwritten so that, when the function returns, the control flow is redirected to execute malicious code, which is called "stack smashing" [3]. Other forms of stack-based buffer overflow attacks overwrite frame pointers and local variables, e.g. function pointers, to affect program behaviors [8, 50]. Many countermeasures against stack-based buffer overflow attacks have been devised, such as StackGuard [17], StackShield [64], Non-executable stack [63] and Libsafe [66], some of which have been widely deployed.

Exploitation of a heap-based buffer overflow is similar to that of a stack-based buffer overflow, except for that there are no return addresses or frame pointers on heap. For some widely deployed memory allocators, such as Doug Lea's malloc [36] for the glibc library, altering memory management information to achieve arbitrary memory overwrites is a general way to exploit a heap-based buffer overflow [11, 33]. More recently non-control data based exploits [12, 13, 15], by means of tampering the content of a memory block adjacent to the overflowed buffer, have been increasing.

As stack-based buffer overflow attacks are better understood and defended, heap-based buffer overflows have gained growing attention of attackers. According to the National Vulnerability Database [43], 177 heap-based buffer overflow vulnerabilities were published in 2009. As of September 2010, 287 heap-based buffer overflow vulnerabilities had been published in that year. Related exploits have affected widely deployed programs [11, 12].

Current buffer overflow detectors can be roughly classified into two categories: static and dynamic approaches. Static analysis tools usually have high false alarm rates; dynamic buffer overflow detectors can provide precise detection and generally there can be no false alarms [71]. However few dynamic heap buffer overflow detectors are widely deployed due to one or more of the following reasons: (1) Most countermeasures result in high performance overhead [1, 5, 9, 22, 27, 34, 52, 68]; (2) Some only protect specific libc functions [5, 19]; (3) A few of them only work with specific memory allocators [2, 19]; (4) Many require source code for recompilation [1, 4, 9, 31, 34, 42, 52]; (5) Some are incompatible with legacy code [4, 31, 42]; and (6) Some require special platforms or hardware supports that are rarely available [1, 34].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

In this paper, we present Cruiser—a novel dynamic heap buffer overflow detector which does not have those limitations. The key ideas are (1) to create a dedicated monitor thread, which runs concurrently with user threads to *cruise* over, or keep checking, dynamically allocated buffers against buffer overflows; and (2) to utilize lock-free data structures and non-blocking algorithms, through which user threads communicate with the monitor thread with minimum overhead and without being blocked. Buffer addresses are collected in a lock-free data structure efficiently without blocking user threads. By traversing the data structure, buffers on heap are under constant surveillance of the concurrent monitor thread. Each dynamically allocated buffer is surrounded by two canary words; as long as a canary is found corrupted, a buffer overflow is detected.

Different from conventional methods that detect buffer overflows inside user threads, which evidently delays protected programs, we propose to move the detection work out of user threads and enforce it in a separate thread, which we call *software cruising*. The approach leverages the hardware evolution trend that multicore processors and multiprocessor machines are more and more popular, which allows us to deploy dedicated monitor threads running concurrently with user threads to enhance application security. However applications may be significantly slowed down due to the overhead of communication and synchronization between the monitor threads and user threads. We address this problem by designing highly efficient lock-free data structures and non-blocking algorithms; their scalability implies the approach can be applied to not only single-threaded programs but also large-scale multithreaded applications. Our experiments show that the software cruising approach is practical—it only imposes an average of 5% performance overhead on SPEC CPU2006, and the throughput slowdown on Apache is negligible on average.

Our contributions include:

- To the best of our knowledge, this is the first work reported in the open literature that utilizes concurrent threads to detect buffer overflows.
- Among existing buffer overflow detectors, this is the first work that utilizes and designs lock-free data structures to support large-scale applications.
- Cruiser can detect buffer overflows that occur in any function, rather than specific libc functions. In addition, Cruiser can detect buffer underflows, duplicate `free`s, and memory leakage.
- Cruiser is legacy code compatible and can be applied to protect binary executables transparently, thus no source code or binary rewriting is needed. In addition, Cruiser does not rely on specific memory allocation algorithms; it can work with any memory allocator. Hence, Cruiser can be directly applied to shrink-wrapped software, and can be deployed easily to large-scale systems such as data centers and server farms in an automated manner.
- We propose a novel concurrent program monitoring methodology called *software cruising*, which leverages multicore architectures and utilizes non-blocking data structures and algorithms to achieve good efficiency and scalability; it can be applied to background security enforcement, as demonstrated in this paper, and concurrent resource reclamation (Section 7).

The remainder of the paper is organized as follows. In Section 2 we briefly discuss lock-free synchronization and review related work. We discuss our motivations in Section 3 and present the design overview in Section 4. In Section 5 we describe the design and implementation. In Section 6 we present the evaluation results. We discuss applications of software cruising beyond buffer overflows in Section 7 and conclude with Section 8.

2. Background

2.1 Lock-free Synchronization

A conventional approach to multithreaded programming is to use locks to synchronize access to shared resources. However, the lock-based approach causes many problems, one of which is lock contention. No matter whether the thread holding a lock is running or descheduled, other threads waiting for the lock are blocked, which limits concurrency and scalability. Another problem is priority inversion, i.e. a low priority thread holding the lock cannot get scheduled while high priority threads are waiting for the lock. Although fine-grained locking reduces lock contention, it introduces more lock overhead and increases the risk of deadlock.

In contrast to the lock-based approach, lock-free and wait-free algorithms allow high concurrency and scalability. An algorithm is lock-free if in a finite number of execution steps, at least one of the program threads makes progress, while an algorithm is wait-free if in a finite number of steps, every thread makes progress [28]. All wait-free algorithms are lock-free but the reverse is not necessarily true. Both are non-blocking and, by definition, they are immune to deadlock and priority inversion.

Lock-free algorithms commonly rely on hardware synchronization primitives. A typical primitive is Compare-And-Swap (`CAS`) [30]; it takes three arguments (`addr`, `expval`, `newval`) and performs the following *atomically*:

```
if (*addr != expval)
    return false;
*addr = newval;
return true;
```

Specifically, if the memory location `addr` does not hold the expected value `expval`, the Boolean `false` is returned; otherwise the new value `newval` is written to it and the Boolean `true` is returned, atomically.

2.2 Related Work

We divide the existing countermeasures against buffer overflow attacks into the following seven categories. Given extensive research in this area, this is not intended to be exhaustive.

Bounds checking: Many static analysis tools fall under this category [20, 69], which detect buffer overflows by examining source code statically and automatically. This approach usually suffers from high false positive or negative rate [71]. Some dynamic approaches [4, 31, 42] change the C pointer representation to carry buffer size information with pointers to enable bounds checking, i.e. "fat pointer," which is incompatible with legacy library code. CRED [52], built on the work of Jones and Kelly [32], does not change pointer representations but associates a buffer bound lookup with each pointer reference. However, the performance overhead is more than 2X. A recent work, baggy bounds checking [2], reduces the cost of bounds lookups by relaxing bounds checking precision, which however may lead to false negative, and it relies on specific memory allocators. Some library-based countermeasures [5, 19, 66] provide bounds checking only for specific functions in the C standard library.

Canary checking: Canary was firstly proposed in StackGuard [17], which tackles stack smashing attacks by placing a canary word before the return address on stack. Attempts to overwrite the return address would corrupt the canary value first. Although there are arguments that canary-based countermeasures can be bypassed [8, 50], the wide deployment and successes of StackGuard and its derivation ProPolice [29] have manifested their effectiveness. Robertson et al. [51] first proposed to use canary to protect heap chunk metadata. A canary is placed at the beginning of each chunk, thus when a buffer on heap is overflowed, the canary of the

adjacent chunk is corrupted, which, however, is not detected until the adjacent chunk is coalesced, allocated or deallocated; therefore the detection relies on program execution.

Return address (RA) shadow stack or stack split: StackShield [64], RAD [14] and their derivations [23, 49] maintain an RA shadow stack, i.e. a copy of the RA is saved on the shadow stack at the prologue of a function call and is compared against the RA on the conventional stack at the epilogue. If the two RAs diverge, a buffer overflow is detected. In [70], the stack is split into an RA stack and a data stack, such that return addresses are protected from buffer overflows.

Non-executable (NX) memory: By setting the memory pages as non-executable, NX memory [63, 65] prevents code injected onto stack and heap from being executed. However it can be bypassed by a return-to-libc attack, which overwrites function pointers or return addresses with function addresses in libc, e.g. `system()`.

Non-accessible memory: Both Purify [27] and Valgrind [68] insert guard zones, which are marked as inaccessible, surrounding dynamically allocated buffers, and track all memory references. When a guard zone access is detected, e.g. due to buffer overflows, an error is reported. Eletric Fence [22] places an inaccessible memory page immediately after (or before) each dynamically allocated buffer. If a buffer is overflowed (or underflowed), a segmentation fault is signaled. Tools in this category result in significant memory and performance overhead.

Randomization and obfuscation: Address Space Layout Randomization (ASLR) [7, 65] randomizes the locations of stack, heap and/or variable locations for each execution, such that a buffer overflow attack, such as return-to-libc, cannot be achieved reliably; that is, probabilistic protection is provided. However as it requires programs to be compiled into position-independent executables, it is incompatible with legacy code. In addition, it may be defeated by brute-force attacks or bypassed by partial overwrite attacks on the least significant bytes of a pointer [21]. PointGuard [16] encrypts pointers stored in memory and decrypts them before loading them into registers, such that pointers corrupted by attackers will not be decrypted to intended values. This countermeasure is incompatible with legacy code and cannot protect non-pointer data. Instead of randomizing pointers, instruction set randomization [6] keeps instructions encrypted, and decrypts them only before they are fetched into processors, which, however, results in substantial performance overhead. It can be bypassed by return-to-libc attacks.

Execution monitoring: Program shepherding [34] monitors control flow transfer in order to enforce a security policy. Buffer overflow attacks that lead to deviant control flow transfer are prevented. Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the control flow adheres to it. Castro et al. [9] proposed to compute a data-flow graph using static analysis and monitor whether the program data flow adheres to the graph. Like Cruiser, n-variant execution [18, 53] also takes advantage of multicore and multiprocessor architectures to enhance security. It runs a few variants of a single program simultaneously; behavioral divergences among the variants raise alarms. Execution monitoring usually imposes high performance overhead.

Despite so many countermeasures, only a few of them, such as StackGuard, ASLR, and NX memory, are widely deployed in production systems. Table 1 summarizes the properties of these three approaches and compares them with Cruiser. The common properties of these approaches include *low performance overhead*, *easiness to deploy and apply*, *no false alarms*,[1] *compatibility with mainstream platforms* and *program semantics loyalty*. In addition to having all the advantages, Cruiser have three other important properties: *compatibility with legacy code*, *no need for recompilation*, i.e. working with binary executables, and *ability to precisely locate corrupted buffers*, which is critical for testing, debugging, and security monitoring.

[1] As described in Section 5.2, one variant of Cruiser does incur false alarms, however, at an extremely low probability ($1/2^{64}$ in 64-bit OS) and can be safely ignored in practice.

	StackGuard	ASLR	NX	Cruiser
Low performance overhead	√	√	√	√
Easy to deploy and apply	√	√	√	√
No false alarms	√	√	√	√
Mainstream platform compatible	√	√	√	√
Program semantics loyalty	√	√	√	√
Legacy code compatible	√		√	√
No need for recompilation			√	√
Able to locate corrupted buffers				√

Table 1. Comparison of some widely deployed countermeasures and Cruiser.

3. Motivations

3.1 Why Concurrent Detection and Challenges

There are mainly two categories of dynamic heap-based buffer overflow detectors. One category [51] detects buffer overflows inside memory allocation functions such as `malloc` and `free`, while the other [5, 19] enforces detection inside specific libc functions such as `strcpy` and `gets`. Both execute detection code inside user threads, which inevitably affects application performance, and the performance overhead is proportionally correlated with the invoke density of related functions. In addition, because detection is enforced in specific functions, they suffer from either severe temporal limitations, i.e. buffer overflows are not detected until one of the `malloc` function family is called, or spatial limitations, i.e. only a few libc functions are protected. Approaches that enforce bounds checking for each buffer reference do not have such limitations; however, they usually incur high performance overhead [32, 52] or false negative rate [2].

We propose to move detection code out of user threads and execute it in a separate monitor thread, which constantly *cruises* over buffers on heap, such that user threads are not delayed. Rich computational resources on modern machines, especially widely available multicore and multiprocessor architectures, enable us to run a dedicated monitor thread without competing too much resources with user threads, in other words, applications can potentially gain enhanced security with no pain.

However, synchronization is one of the major challenges. In Cruiser, a collection of heap buffer addresses needs to be maintained, so that the monitor thread surveils live buffers, and in the meanwhile avoids checking deallocated buffers, which would otherwise incur false alarms or segfaults. Therefore, the user threads and monitor thread have to be synchronized when buffers are allocated or deallocated. A conventional approach to achieving synchronization is to use locks; however, it has various limitations, such as severe performance degradation due to lock contention and low scalability, which is manifested by our first attempt.

In our first attempt, a lock-based red-black tree was used to collect buffer addresses. Inside a `malloc` call, the address of the newly allocated buffer is inserted into the tree with $O(\log n)$ time complexity where n is the number of collected addresses. Similarly inside a `free` call, the address of the released buffer is removed from the tree with $O(\log n)$ complexity also. Meanwhile a monitor thread traverses the tree to check the buffers. All the buffer address insert, delete and traverse operations are synchronized using locks. Our experiments showed that user threads were significantly

delayed. The problem becomes more severe as more user threads contend locks and the tree grows.

3.2 Why Lock-free and Challenges

The limitations of lock-based approach pushed us towards lock-free synchronization in order to avoid lock contention and improve scalability. However, the difficulty of designing non-blocking algorithms is well recognized, which often thwarts the application of this approach.

We escaped the problems in our second attempt by utilizing the state-of-the-art extensible lock-free hash table algorithm proposed by Shalev and Shavit [62], such that the user threads and monitor thread can operate on the hash table concurrently, and each buffer address can be inserted into or removed from the hash table in $O(1)$ time. Although good scalability is achieved, the operation time is significant compared to malloc and free calls. Specifically, the slowdown of each pair of malloc and free calls observed in our experiment is more than 5X on average. The overhead is unacceptable for many applications with massive dynamic memory allocation.

To address these challenges, we have designed our own lock-free data structures and non-blocking algorithms to achieve concurrent detection with low overhead and high scalability, which will be presented in Section 4 and 5.

4. Design Overview

In addition to custom lock-free data structures, two design choices were made. First, as presented above, removing buffer addresses inside free calls may significantly delay user threads. In Cruiser the free function marks the buffer with a tombstone flag; when the monitor thread checks the buffer and finds it no longer alive, the monitor thread removes the buffer address from the collection of heap buffer addresses, such that the concern of delaying frees is resolved and the data structure representing the buffer address collection can be simplified. The details are covered in Section 5.2. Second, instead of modifying a specific memory allocator, Cruiser is implemented as a dynamic shared library to interpose the malloc function family and it passes the allocation requests to the corresponding memory allocator functions, therefore Cruiser can work with any memory allocator and it can be applied to protecting binary executables without instrumenting them.

4.1 Buffer Structure

Our method inserts two canary words around each buffer, namely *head canary* and *tail canary*, as shown in Figure 1, so that whenever a buffer is overflowed (underflowed), the tail (head) canary is corrupted. The *size* field, which is the encryption (XOR) result of the buffer size and a secret key, is used to locate the tail canary given a buffer address. As buffer size information is encrypted, it is not leaked to attackers, and it is more difficult for attackers to counterfeit. The head canary is the encryption result of another secret key, the buffer size and the buffer address. If the head canary and the size field cannot be decrypted to consistent size values, a buffer overflow is detected. As the buffer address is used to generate the canary, each buffer has a unique head canary, thus even if the canary of a buffer is leaked, it is difficult for attackers to forge the canary of another buffer without knowing the buffer sizes and addresses. The tail canary is encrypted and verified the same way using a different secret key. All the keys are initialized as random numbers when the monitored program is started.

4.2 Cruiser

Our previous attempts maintain a collection of buffer addresses but lead to high overhead. To efficiently collect memory allocation information, we design the *cruiser information collection* (CIC) architecture which is composed of (1) a lock-free *express* data structure onto which user threads put information, (2) a lock-free *warehouse* data structure that supports multiple threads to concurrently insert, delete and access information, and (3) a non-blocking *deliver* thread to copy the information from the express to the warehouse data structure. Instead of inserting information into the warehouse directly, user threads put the information onto the express data structure highly efficiently, and the deliver thread takes care of the rest of the information collection work, thus performance impact on user threads is minimized. In addition, as the warehouse structure supports concurrent operations, CIC scales well.

Figure 1. Buffer structure.

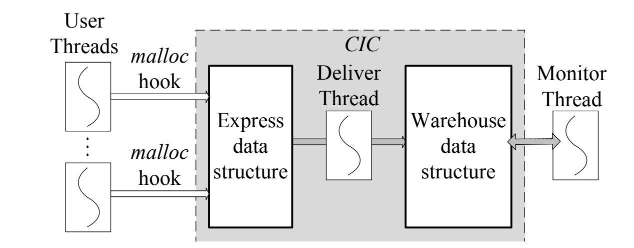

Figure 2. Cruiser architecture.

Based on CIC, we present a dynamic heap-based buffer overflow detector—Cruiser, which uses CIC to collect memory allocation information, e.g. buffer addresses and sizes. As shown in Figure 2, the malloc calls are hooked to place the buffer allocation information onto the express data structure and return promptly; the deliver thread then finishes the information collection.

From the perspective of Cruiser, the life cycle of a dynamically allocated buffer can be divided into three phases: **Pre-checking**: Inside a malloc call, a buffer that is three words larger than what the user thread requests is allocated. The buffer is filled as specified in Section 4.1, and the buffer allocation information used for overflow detection, such as the buffer address, is put onto the express data structure. Then the malloc call returns the address of the *user buffer* (see Figure 1). **Checking**: The pre-checking phase ends when the deliver thread moves the address from the express to the warehouse data structure, which is traversed by the monitor thread to detect buffer overflows. **Post-checking**: Inside the free call, the buffer is marked with a tombstone flag by encrypting the head canary once again using another key; later when the monitor thread checks the buffer and finds it no longer alive, the dated buffer information is removed from the warehouse by the monitor thread, so it is the monitor thread rather than user threads that tides up dated metadata information.

5. Design and Implementation

This section describes the design and implementation of Cruiser. Section 5.1 describes the data structures in CIC and how CIC is used in Cruiser to maintain buffer information. Section 5.2 presents the algorithms to release buffers and delete dated metadata information. We elaborate special issues on extensions and optimizations in Section 5.3.

5.1 Collection of Buffer Information

5.1.1 Express data structure

We implement the express data structure based on the single-producer single-consumer FIFO wait-free ring buffer proposed by Lamport [35]. Lamport's algorithm allows a producer thread and a consumer thread to operate concurrently on a ring. The synchronization overhead between the producer and the consumer is low, as two threads are synchronized via read/write instructions on the two control variables *head* and *tail*. Because of its high efficiency, the data structure has been applied to Gigabit network packet processing systems [24, 37].

To avoid the failure of Enqueue operation when the ring is full, we extend the basic ring to a linked list of rings, called *CruiserRing*, as shown in Figure 3. Whenever the ring is full and a new element is produced, instead of returning failure in Enqueue as in Lamport's algorithm, the producer creates a new ring with doubled capacity and links it after the full ring; the producer proceeds to insert elements into the new ring. Accordingly, in Dequeue when the ring is consumed up and another ring is linked after it, the consumer destroys the empty ring and proceeds to work on the next one. Because the ring size grows exponentially, as long as the speed of the consumer matches that of the producer, CruiserRing will converge to a stable state quickly. (The speed mismatch problem is addressed in Section 5.3.) Unless the new ring creation fails, CruiserRing ensures the success of the producer, which implies that the producer always moves on without dropping data.

As each CruiserRing supports one producer thread, a CruiserRing is needed for each producer thread. The method AddCruiserRing (see Figure 3) shows how to construct a list of CruiserRings in a lock-free manner, such that a single consumer thread can walk along the list to access all CruiserRings.

5.1.2 Warehouse data structure

We implement the warehouse data structure as a custom lock-free list, called *CruiserList*. CruiserList is a linked list of *segments*, each of which is a linked list itself with a never-removed dummy node as the segment head, as shown in Figure 4 and 5.

The basic form of CruiserList contains one segment, which supports a single *insert* thread to insert nodes and a single *traverse* thread to traverse the list concurrently. The method CheckNode is invoked in Traverse to check each node and returns whether the node should be deleted.

New nodes are always inserted between the dummy node and the first genuine node (the node linked immediately after the dummy node). If the first genuine node is determined to be deleted (Line 63), it should not be removed directly, as it may otherwise lead to list corruption or node loss. Specifically, if the first genuine node is removed between the execution of Line 51 and Line 52 when a new node is being inserted, the list is corrupted, because the newly inserted node has been linked to a deleted node. Another situation is when the first genuine node M is determined to be deleted, a new node N is inserted, which is not known by the traverse thread. Consequently, by removing node M, the dummy node is linked to the node after node M, such that node N is lost. The contention between node insertion and deletion is a common problem in lock-free data structures.

A conventional method to resolve the contention problems in non-blocking algorithms is to use CAS in a loop to insert or delete a node, as in the CruiserRing method AddCruiserRing (see Figure 3). However, CAS is relatively expensive and due to contention concurrent operations may experience frequent failure and retry of CAS instructions, which delays the progress of concurrent threads.

In CruiserList, we essentially eliminate the contention and thus CAS is not needed, as shown in the method Traverse. In our algorithm, the first genuine node is never removed until new nodes have been inserted, thus new nodes can always be inserted between the dummy node and the first genuine node safely by the insert thread, while the traverse thread never touches the link between the dummy node and the first genuine node. Therefore, the node insertion and deletion operations essentially play in different arenas, and thus have no contention.

Specifically, when the first genuine node is determined to be deleted, it is marked as to-be-deleted by calling the method MarkDelete, which fills a special *null* value in the data field of the node, or sets the least significant bit (LSB) of the *next* pointer of the node, as the node address is usually word-aligned and the LSB of

```
1  struct Ring {
2      Element *buffer;
3      unsigned int size;
4      unsigned int head, tail;
5      Ring *next; // next Ring
6  };
7
8  struct CruiserRing {
9      Ring *pr, *cr; // producer ring and consumer ring
10     CruiserRing *next; // next CruiserRing
11 };
12
13 CruiserRing *Head; // head of CruiserRing list
14
15 NEXT(index, size) { return (index + 1) % size; }
16
17 Enqueue(pr, data) {
18     if (NEXT(pr−>head, pr−>size) == pr−>tail) {
19         newRing = createRing(2 * (pr−>size));
20         if (null == newRing)
21             return failure;
22         pr−>next = newRing;
23         pr = newRing;
24     }
25     pr−>buffer[pr−>head] = data;
26     pr−>head = NEXT(pr−>head, pr−>size);
27     return success;
28 }
29
30 Dequeue(cr, data) {
31     if (cr−>head == cr−>tail) {
32         if (null == cr−>next)
33             return failure;
34         temp = cr; cr = cr−>next;
35         destroy(temp);
36         return Dequeue(cr, data);
37     }
38     data = cr−>buffer[cr−>tail];
39     cr−>tail = NEXT(cr−>tail, cr−>size);
40     return success;
41 }
42
43 AddCruiserRing(cruiserRing) {
44     do {
45         cruiserRing−>next = oldValue = Head;
46     } while (!CAS(&Head, oldValue, cruiserRing));
47 }
```

Figure 3. CruiserRing (Express data structure).

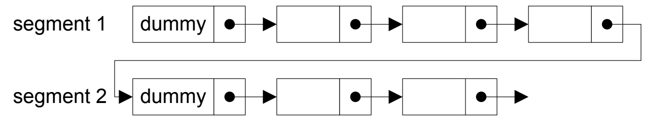

Figure 4. CruiserList.

```
48 Node *head; // head of CruiserList
49
50 Insert(dummy, node) {
51     node−>next = dummy−>next;
52     dummy−>next = node;
53 }
54
55 Traverse() {
56     Node *prev, *cur, *next;
57     cur = leftBoundary−>next;
58     if (cur == null)
59         return;
60
61     /*Process the first genuine node*/
62     if (!IsMarkedDelete(cur))
63         if (CheckNode(cur) returns PLEASE_DELETE_ME)
64             /*Node removal is deferred to avoid contention*/
65             MarkDelete(cur);
66
67     /* Process the rest genuine nodes */
68     prev = cur; cur = cur−>next;
69     while (cur != rightBoundary) {
70         next = cur−>next;
71         if (IsMarkedDelete(cur) ||
72         CheckNode(cur) returns PLEASE_DELETE_ME) {
73             prev−>next = next;
74             DeleteNode(cur);
75         }
76         else
77             prev = cur;
78         cur = next;
79     }
80 }
81
82 AddSegment() {
83     Node *newDummy = AllocateDummyNode();
84     do {
85         newDummy−>next = oldValue = head;
86     } while (!CAS(&head, oldValue, newDummy));
87     return newDummy;
88 }
```

Figure 5. CruiserList (Warehouse data structure).

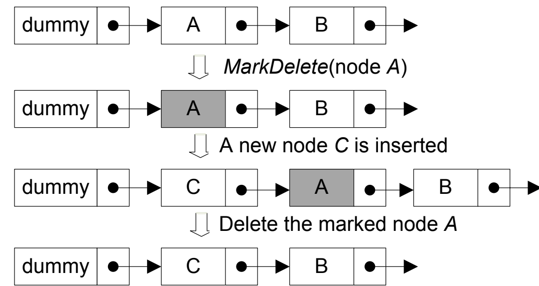

Figure 6. Deletion of the first genuine node.

the *next* pointer is thus not used. Then the marked node is removed in a future round of traverse when it is no longer the first genuine node. Figure 6 shows the process of removing the first genuine node A. It is first marked, but not deleted. After a new node C is inserted, it will be deleted shortly. It is possible that no more new nodes are inserted and the marked node sticks in the list; however, there is only one such node in a segment and normally this occurs only in the residual period of program execution. Note that the user buffer is freed; only the first metadata node may stay alive after the corresponding buffer is released. We can resolve this using CAS if it becomes a serious issue. The Insert method inserts a node, just as in a single-threaded list, between the never-removed dummy node and the first genuine node which is never removed directly.

The technique of marking a node as to-be-deleted was first used in the lock-free FIFO queue algorithm [48] proposed by Prakash et al., then used in Harris's [26] and Michael's [40] lock-free lists, respectively. All of them use this technique to prevent new nodes from being linked to a marked node. As insertion and deletion may operate on the same node, contention still exists; and they rely on CAS. Our algorithm allows new nodes to be linked to a marked node. Only simple reads and writes are needed.

The basic CruiserList can be easily extended to multiple segments using the method AddSegment, so that it can support multiple insert and traverse threads. Each insert thread has a thread-private variable pointing to the dummy node of a different segment, into which this thread inserts nodes. On the other hand, the segments can be partitioned into multiple disjoint groups; each traverse thread walks on a different group denoted by two thread-private variables *leftBoundary* and *rightBoundary*, which point to the dummy nodes of segments. If there is only one traverse thread, its segment group consists of the whole CruiserList. For the sake of simplicity, the Traverse method in Figure 5 is only for one-segment groups. The time for a traverse thread to cruise through its segment group once is called a *cruise cycle*.

Compared to general-purpose lock-free lists, CruiserList is highly efficient and has the following advantages: (1) *Wait-free access and zero-contention*: Both insert and traverse threads keep making progress, and node insertion and deletion are executed in different arenas; (2) *No ABA problem*: The ABA problem [30] is historically associated with CAS. It happens if in a thread a shared location with a value A was read, then CAS comparing the current value of the shared location against A succeeds though it should not, as between the read and the CAS other threads change the value of the shared location from A to B and back to A again. The only CAS in Line 86 has no ABA problem, because it is impossible for *head*, which was changed from the address of the dummy node A to that of the dummy node B, to change back to A without removing any dummy nodes; and (3) *No special memory reclamation needed*: For a typical lock-free data structure, when a node is removed by a thread, its memory cannot be released immediately because other threads may be accessing it. So special memory reclamation mechanisms are needed, such as reference counters and hazard pointers [39]. On a given segment of the CruiserList, the traverse thread is the only thread that deletes nodes, so it is not concerned with accessing nodes being released by other threads. It is not a problem for the insert thread either, as it only accesses the content of the dummy node, which is never removed.

5.1.3 Applying CIC in Cruiser

Cruiser uses the CIC mechanism to collect memory allocation information (see Figure 2). The malloc calls of user threads are hooked and memory allocation information is put on the CruiserRings. The deliver thread moves the information from CruiserRings to CruiserList, while the *monitor* thread calling Traverse of CruiserList cruises over buffers to detect overflow according to the collected information in CruiserList. The buffer overflow detection code is executed in CheckNode (see Figure 5), which returns PLEASE_DELETE_ME if the buffer is found no longer alive. More details about CheckNode are described in Section 5.2.

Cruiser is implemented as a dynamic shared library to interpose the malloc function family. It contains a constructor (initialization) function, which gets executed when the monitored program is started. Inside the constructor function, the keys are initialized with random numbers in /dev/urandom, the CruiserList is created and initialized, and the deliver thread and the monitor thread are created, which share the same address space as user threads. Each

user thread creates its CruiserRing when it invokes its first malloc call. All the memory blocks used by Cruiser (the data structures and keys) are allocated using mmap with two inaccessible guard pages [22] surrounding each of them, such that they cannot be overflowed.

5.2 Buffer Release and Node Removal

Once a buffer is freed, the node in the CruiserList containing the corresponding buffer information becomes dated and should be removed; otherwise, buffer overflow checks over the buffer may incur false alarms or segmentation faults, as the buffer memory may have been reused or unmapped. Removing dated nodes inside free calls may significantly delay user threads. We address the problem with the following two approaches; both enable the monitor thread to tidy-up CruiserList.

The first approach is a lazy two-step memory reclamation algorithm. First, when a free call is intercepted, the target buffer is marked with a tombstone flag by encrypting the head canary with another key, called the *release key*; the free call returns without releasing the buffer, which becomes a *zombie buffer*. Second, when the monitor thread checks the buffer and finds it marked with a tombstone, the method CheckNode in Figure 5 releases the buffer and returns PLEASE_DELETE_ME. This approach removes dated information effectively without incurring false alarms or segfaults. The drawback is that buffer release is delayed; however, since all zombie buffers are bound to be released no later than the next cruise cycle, the delay should be reasonably short.

The second approach does not delay memory reclamation; however, it requires some changes about the head and tail canaries and needs the assistance of recovery techniques. The head canary field is filled with a random number (rather than the encrypted size value), while the tail canary is the encryption (XOR) result of the head canary value and a key. The random number along with the buffer address is collected in CruiserList. Inside the free call, the target buffer is checked against buffer overflow by decrypting the tail canary and comparing with the head canary.[2] If they are different, a buffer overflow is detected; otherwise the head canary is set as zero, after which the buffer is released and the free call returns. When the monitor thread checks a buffer and finds the number stored in the head canary is not the same as that stored in the CruiserList, it assumes the buffer has been released and then the corresponding dated node is removed from the CruiserList. After a buffer is released and reused, the memory location of the original head canary may happen to be written with the same value as that before the buffer was released, so that when the monitor thread checks the node using the dated buffer information, it would incorrectly determine this buffer is still alive and thus false alarms are possible; however, the probability is extremely low (in 64-bit OS, it is $1/2^{64}$), which can be safely ignored in practice.

Three scenarios need to be considered for the second approach. First, when a buffer is tampered due to an overflow occurred in its preceding adjacent buffer, it would be incorrectly determined as a released buffer by the monitor thread; however, this does no harm and as the adjacent buffer is under surveillance, the overflow can still be detected. Second, when a buffer is underflowed, the monitor thread would also treat it as a released buffer. Underflows are rare compared to overflows; moreover we can address the problem by saving the buffer size in the CruiserList node, against which the size field of each buffer is checked by the monitor thread in CheckNode to detect underflows.

[2] If the size field has been corrupted, the tail canary cannot be located correctly. As a result, the read of the tail canary may incur segfault, which, however, essentially exposes buffer overflows. If necessary, the same recovery technique described here can be used to deal with segfault.

Third, when the monitor thread checks a buffer that has been released and the corresponding memory page(s) has been unmapped, a segfault is triggered. The problem can be addressed using some recovery techniques [47]. In Linux, a SIGSEGV signal handler can be installed firstly. Each time before the monitor thread accesses a buffer, it calls sigsetjmp to save the calling environment. Once a SIGSEGV signal is triggered due to an invalid access, the monitor thread is trapped to the SIGSEGV handler, and the calling environment can be recovered by calling siglongjmp. Windows also has similar recovery mechanism called Structured Exception Handling [41].

Cruisers with the two approaches are called *Lazy* Cruiser and *Eager* Cruiser, respectively. We have implemented both in Linux.

5.3 Extensions and Optimizations

5.3.1 Extensions

Flexible deployment options: The deployment of Cruiser is flexible. One method is to implement Cruiser as a dynamic shared library. By setting the LD_PRELOAD environment variable to the path of the Cruiser library, an administrator can selectively enable Cruiser for certain applications. With this method the malloc function family are interposed by Cruiser, which invokes the memory allocation functions in the system library to enforce dynamic memory allocation, thus no system library is altered. This is the deploying method we adopt in the experiments.

A second method is to integrate Cruiser with the system dynamic library for dynamic memory allocation. The advantage is that memory and performance overhead can be reduced. For example, the overhead due to malloc function family interposition can be avoided; some memory allocators, such as dlmalloc [36], place the buffer size information in the beginning of each chunk, so Cruiser does not need to maintain that information additionally.

A third method is to implement Cruiser as a static library and integrate it into the compiler. Considering the two methods above cannot be applied to statically linked applications, the third method is complementary to them. Regardless of the deploying method, Cruiser has no effect on applications that perform their own memory management, neither can it detect buffer overflows inside a structure currently, which is a limitation shared by other techniques that detect buffer overflows at the level of memory blocks [2, 5, 19, 22, 27, 51, 68]. However, Cruiser can be extended to monitor buffers inside a struct by inserting canary words.

More Cruiser threads: Although our experiments show that the Cruiser configuration with one deliver thread and one monitor thread is sufficient for common applications, it may be desirable to extend Cruiser with multiple deliver and monitor threads. For example, there may be many user threads requesting dynamic memory intensively that a single deliver thread cannot match the speed of buffer allocation; or the CruiserList is so long that it takes much time for a monitor thread to traverse through the CruiserList once. As both CruiserList and the list of CruiserRings support multiple threads, Cruiser can be easily extended with more cruiser threads to protect various applications.

5.3.2 Optimizations

Memory reuse: To mitigate memory allocation intensity and speed up node insert and delete in CruiserList, a ring buffer is adopted to store the addresses of removed nodes with the monitor thread as the producer and the deliver thread as the consumer. Nodes removed from CruiserList are not deleted but stored in the ring unless it is full. Accordingly when the deliver thread needs a node, it first tries to retrieve a node from the ring; only when the ring is empty, a new node is allocated. The simple ring buffer can be replaced with other advanced wait-free queues, such as a CruiserRing, to support more efficient nodes buffering strategy.

On the other hand, each user thread requesting dynamic memory owns a CruiserRing. Considering some applications fork and kill threads frequently, instead of allocating and releasing CruiserRings intensively, we reuse CruiserRings. A Boolean flag indicating whether the CruiserRing is available for reuse is added into the CruiserRing structure. As the deliver thread traverses along the list of CruiserRings, if it detects a user thread has exited, it marks the related CruiserRing as available for reuse. When a user thread needs a CruiserRing, it will try to reuse an available CruiserRing before allocating a new one.

Backoff strategies: Although our experiments show that Cruiser imposes low overhead, the performance can be further improved with backoff strategies, for example, by inserting *NOP* instructions or `sleep` calls inside the monitor thread. Reduced monitor intensity leads to less memory access interference, thus decreases performance impact. Cruiser can also switch to monitor buffers selectively, for example, buffers involved in data flows stemming from networks or user inputs. For Lazy Cruiser, it is a good choice to ignore large buffers and hence release them inside `free` calls directly under intense memory pressure. More advanced monitor strategies based on computational resource dynamics can be adopted as well.

Variants of the deliver thread: In Cruiser, the deliver thread is busy polling CruiserRings. Actually it can go to sleep when there is no information to deliver and be waken up by user threads via signals. To avoid sending a signal per `malloc` call, a global status flag indicating whether the deliver thread is asleep can be used. The flag is set as awake or asleep by the deliver thread; a wake-up signal is sent to the deliver thread only when it is asleep.

Another variant is to combine the deliver thread and the monitor thread; we can have the hybrid thread delivering information and monitoring nodes alternatively, such that only one busy thread is needed and the data structures can be further simplified. The drawback is that the monitoring may be interrupted frequently.

Better ring algorithms: Based on Lamport's ring [35], some other ring algorithms have been proposed [24, 37]. We used the ring algorithm proposed by Lee et al. [37] in our experiments to mitigate the false sharing problem in Lamport's ring.

6. Evaluation

We evaluated the effectiveness of Cruiser, its performance and memory overhead, and analyzed the detection latency issue. This section presents our results.

6.1 Effectiveness

We evaluated the effectiveness of Cruiser with two experiments. Our first experiment was carried out using the SAMATE Reference Dataset (SRD) [44] maintained by NIST. The dataset contains 12 test cases on heap-based buffer overflows due to contiguous writes, which are caused by assignments, `memcpy`, `strcpy`, `snprintf`, etc., and another 10 test cases that fix the overflows. Cruiser detects all the overflows in the 12 test cases and there is no false positive for the 10 sound test cases.

The second experiment tested the effectiveness against both well-known historic exploits (wu-ftpd [57], Sudo [61], CVS [55]) and recently published vulnerabilities (libHX [58], Lynx [59], Firefox [56]), shown in Table 2. Each vulnerable program was run with Cruiser and attacks were launched on them. Each attack was executed 50 times and Cruiser detected all the overflows, duplicate and invalid `free`s.

These experiments demonstrate that our technique is effective in detecting not only heap-based buffer overflows but also memory leakage and heap corruption including duplicate `free`s and `free`s on invalid pointers. Therefore, Cruiser is a good candidate for finding heap corruption and memory leakage defects during development as well as monitoring production systems.

Program	Vulnerability
wu-ftpd 2.6.1	Free calls on uninitialized pointers
Sudo 1.6.4	Heap-based buffer overflow
CVS 1.11.4	Duplicate free calls
libHX 3.5	Heap-based buffer overflow
Lynx 2.8.8 dev.1	Heap-based buffer overflow
Firefox 3.0.1	Heap-based buffer overflow

Table 2. The effectiveness experiment against real-world vulnerabilities.

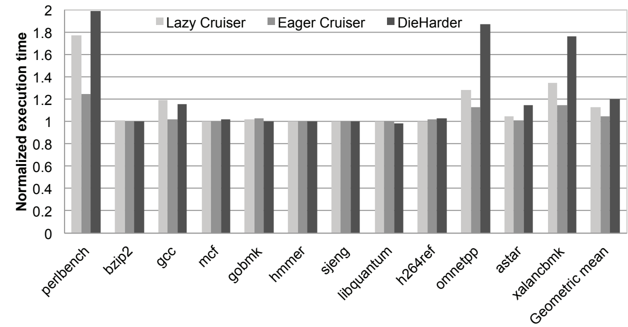

Figure 7. Execution time with Cruiser (normalized by the execution time without Cruiser), compared with DieHarder. The last set of bars is the geometric mean of the execution time of all benchmarks.

6.2 Performance and Memory Overhead

We evaluated the performance and memory overhead of Cruiser using the SPEC CPU2006 Integer benchmark suite. The experiments were performed on a Dell Precision Workstation T5500 with two 2.26GHz Intel Xeon E5507 quad-core processors and 4GB of RAM running 32-bit Linux 2.6.24. Cruiser was implemented as a dynamically linked library and loaded by setting the LD_PRELOAD environment variable. In all the experiments, Cruiser created one deliver thread and one monitor thread with the optimizations *memory reuse* and *better ring algorithms* enabled. We ran each experiment three times and present the average result. The variance was negligible.

We evaluated both Lazy Cruiser and Eager Cruiser, and compared with DieHarder [45], which also provides probabilistic heap safety. Figure 7 shows the execution time of the two implementations normalized by the execution time of original programs. The average performance overhead is 12.5% for Lazy Cruiser and 5% for Eager Cruiser, while DieHarder imposes 20% penalty on average. For the majority of the benchmark programs, the overhead imposed by Cruiser is negligible. The perlbench has the highest overhead due to its significantly dense dynamic memory allocation. Eager Cruiser performs generally better than the lazy version, mainly because the former allows immediate memory reclamation and reuse. The experiments show that Cruiser can be deployed in field practically.

In addition to the three-word tag associated with each buffer, the memory overhead of Cruiser is mainly due to its data structures CruiserRings and CruiserList. For Lazy Cruiser, zombie buffers are another source. To precisely analyze the memory overhead, as shown in Table 3, we measured the maximum size of CruiserRing and the maximum and average lengths of CruiserList normalized by the live buffer counts at sample time (we sampled at the end of each cruise cycle), respectively, from which we can get the percentage of dated nodes and zombie buffers. As the CruiserList length is normalized, the maximum length can be less than the average

Bench-mark	Maximum CruiserRing size	Maximum CruiserList length	Average CruiserList length	Average cruise cycle (μs)
perlbench	8192 (1024)	1.06 (1.16)	1.02 (1.06)	4.3e4 (1.2e5)
bzip2	1024 (1024)	1.00 (1.00)	1.00 (1.00)	.43 (1.2)
gcc	2048 (2048)	1.02 (1.05)	1.00 (1.01)	1.3e3 (3.5e3)
mcf	1024 (1024)	1.00 (1.00)	1.00 (1.00)	.35 (.59)
gobmk	1024 (1024)	1.00 (1.00)	1.00 (1.00)	.37 (1.6)
hmmer	1024 (1024)	1.11 (1.29)	1.00 (1.00)	26 (1.6e2)
sjeng	1024 (1024)	1.11 (1.00)	1.00 (1.00)	.36 (0.82)
libquantum	1024 (1024)	1.00 (1.00)	1.00 (1.00)	.16 (0.49)
h264ref	1024 (1024)	1.00 (1.00)	1.00 (1.00)	3.3e2 (1.7e3)
omnetpp	2048 (1024)	1.08 (1.08)	1.02 (1.08)	9.8e4 (3.2e5)
astar	4096 (2048)	1.04 (1.06)	1.00 (1.00)	19 (88)
xalancbmk	2048 (1024)	1.00 (1.00)	1.02 (1.07)	7.3e4 (1.5e5)

Table 3. Memory overhead and cruise cycle (Results of Eager Cruiser are enclosed in parentheses; the maximum/average CruiserList lengths are normalized by the maximum/average heap buffer counts, respectively).

one, as in the case of xalancbmk. The initial CruiserRing has 1024 elements; the size of each element is one word in Lazy Cruiser and two words in Eager Cruiser, respectively. For the majority of benchmarks, the CruiserRing does not grow, while the maximum CruiserRing in perlbench test with Lazy Cruiser experienced 3 times of growth (recall that the ring grows exponentially). The length of CruiserList is close to the count of live buffers on heap. In other words, the percentage of dated nodes and zombie buffers is low, and on average it is negligible.

6.3 Scalability

To evaluate the scalability of our approach on multithreaded programs, we compared the throughputs of the Apache web server with and without Cruiser. We ran Apache 2.2.8 on the same workstation specified in Section 6.2, and used ApacheBench 2.3, which ran on a machine with a 2.4GHz Intel Core 2 Duo processor, 4GB of RAM, and Mac OS X 10.6.4, to measure the Apache throughput over a Gbit LAN network. We issued repeated requests for a 5KB HTML page with various numbers of concurrent clients. We observed that Apache allocated two heap buffers per request; ApacheBench issued one million requests for each concurrency number. Figure 8 shows the throughputs of Apache (labeled as baseline) and Apache with Lazy Cruiser and Eager Cruiser, respectively. The throughputs of Apache with the two versions of Cruiser are almost the same. The maximum 3% slowdown appears around concurrency number 7, while the average slowdown is negligible. For concurrency numbers greater than 11, the throughputs with and without Cruiser are almost identical. We measured until concurrency number 110 when the client's CPU was saturated, and the throughput slowdown remained negligible. The machine running Apache has 8 cores in processors, thus the Cruiser threads compete processor time since concurrency number 6; however as the working threads in Apache increase, the percentage of processor time used by the Cruiser threads decreases, and thus the slowdown declines.

6.4 Detection Latency

Heap-based buffer overflows can be divided into two classes [51]. One class of attacks alter memory allocators' metadata. Exploits in this class are often achieved by releasing a corrupted buffer [33]. Cruiser defeats this kind of exploits completely, as all buffers are checked before release.

The other class comprises attacks that overflow a buffer to alter the content of its adjacent memory block. This kind of exploits

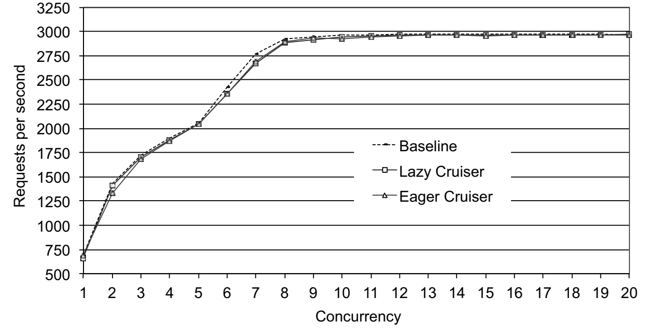

Figure 8. Throughtputs of the Apache web server for varying numbers of concurrent requests.

are not achieved until the corrupted content is used; the attacks are not detected until corrupted buffers are checked by a monitor thread. Like DieHarder, Cruiser provides probabilistic heap safety for this kind of attacks. Assume an attack takes time E to achieve the exploit after canary smashing and a cruise cycle takes time C, if $E \geq C$, Cruiser can detect the overflow before the exploit is achieved; otherwise, assume the detection latency (elapsed time since the overflow until detection) is uniformly distributed on the interval $(0, C]$, the probability P for Cruiser to detect an exploit before it completes is E/C. As shown in Table 3, with Lazy Cruiser, 5 of the 12 benchmarks' average cruise cycles are shorter than 0.5μs, and 8 of them are not longer than 0.5ms; with Eager Cruiser, 5 benchmarks' average cruise cycles are not longer than 1.6μs, and 7 of them are shorter than 0.2ms. The average cruise cycles for Apache test are 16μs and 78μs for Lazy and Eager Cruisers, respectively. Considering $C = NT$, where N is the number of nodes in the CruiserList and T is the average time for a monitor thread to check a node, we can expect a high prevention probability by keeping N small. One way is to divide the CruiserList into several segment groups and create the same number of monitor threads, each of which cruises over a shorter part of CruiserList. For example, by dividing the CruiserList into two parts and running one more monitor thread on either part in omnetpp test, the average cruise cycle decreased by 42% and 47% for Lazy Cruiser and Eager Cruiser, respectively. Another way is to only monitor suspicious buffers, for example, those buffers involved in data flows that stem from networks or user inputs.

Considering Cruiser shares the same address space as user programs, an attacker with arbitrary memory access privileges of the compromised program can bypass Cruiser theoretically. Otherwise a reliable and precise attack against Cruiser is hard to build. In Cruiser, the canaries of a buffer are the XOR result of the buffer's address, size and the keys which were initialized using random numbers, so it is difficult to predict or restore a canary. For example, an elaborate attack that exploits other vulnerabilities, such as format string [50], to obtain the keys still needs the target buffer size and address information to calculate its canaries. Blind access for Cruiser's data structures will normally incur segfault, as they are surrounded by inaccessible guard pages. The function `pthread_kill` can be interposed by Cruiser to prevent Cruiser threads from being killed, and Cruiser threads can detect the liveness of each other when running, thus it is very unlikely to subvert Cruiser. Detection can be evaded by terminating the process, which, however, explicitly exposes the attack and is not a usual way of real attacks.

7. Software Cruising

We have applied the software cruising technique to efficient heap overflow detection by moving the detection work out of user

threads. Potentially all the *inlined* verification, monitoring and resource reclamation work can be migrated from user threads to one or more monitor threads for concurrent execution in *background*. This section discusses other applications of software cruising, some of which we are currently working on as an extension of this work.

Background Software Monitoring: Depending on the security policies enforced, inlined security enforcement may incur high performance overhead. Software cruising takes a very different way by moving inlined security enforcement out of user threads and executing them in concurrent monitor threads, which can reduce considerable performance overhead. Although synchronization and race conditions between user and monitor threads are potential challenges, they can be solved using lock-free data structures as in Cruiser. For example, we can implement a call trace collector by instrumenting the call instruction in the binary and placing the target address in a CruiserRing. A concurrent monitor thread analyzes the call trace to evaluate whether specific control-flow policies are followed. As simple examples, some control transfers are suspicious; or a control-flow policy may require that a certain function is called no more often than another function (such restrictions may be desirable to prevent some "confused deputy" attacks [25]). The monitor thread can detect abnormalities based on the call trace and a finite automaton or simple counting. Other straightforward applications include integrity-checking of important program structures such as the Global Offset Table.

OS Kernel Cruising: It is desirable to adopt software cruising to monitor OS kernel memory integrity and other safety and liveness properties. We plan to develop a prototype that can monitor integrity of OS kernel memory. One of the challenges of cruising OS kernels is how to minimize the impact on the kernel memory layout since kernel code contains many low-level programming idioms that rely on certain memory layouts. One way to solve this problem is to selectively monitor some buffers and make manual transformation.

Concurrent Resource Reclamation: Software Cruising can also be applied to implement efficient resource reclamation, for example safe memory reclamation for lock-free data structures. For lock-free dynamic objects, when a thread removes a node, it is possible that some other thread has earlier read a reference to that node, and is about to access its contents, therefore the memory occupied by the node should not be released or reused directly. When designing lock-free data structures, safe memory reclamation is a major concern. Recent progress was made by Michael [39]. The core idea is to associate a number of pointers, called *hazard pointers*, with each thread. A hazard pointer points to a node that may be accessed later by that thread; whenever a thread frees a retired node, it has to scan hazard pointers of other threads to make sure the node is not pointed to by any of the hazard pointers. To achieve a low amortized overhead, a thread does not free retired nodes until it accumulates a certain number of retired nodes. The inlined batch processing of retired nodes inevitably delays user threads and memory reclamation. The problem can be solved elegantly by deploying a concurrent thread, which takes over the work of memory reclamation by scanning hazard pointers to determine which retired nodes can be released safely. The original methodology is complementary to this solution in case too many retired nodes are accumulated.

8. Conclusion

We have introduced a novel technique, *software cruising*. The core idea is to mitigate inlined verification, monitoring and resource reclamation work from user threads to a concurrent monitor thread. Through lock-free data structures and non-blocking algorithms the monitor thread and user threads can be synchronized with low overhead and high scalability.

We have applied this technique to Cruiser, a dynamic heap-based buffer overflow detector on the Linux platform. It is straightforward to adapt Cruiser to other platforms such as Windows and Mac. We have evaluated Cruiser on a variety of programs to show its effectiveness. The performance overhead of monitoring SPEC CPU2006 benchmark is about 5% on average, and negligible in the majority of cases. Cruiser also scales well on multithreaded programs; the slowdown on the Apache throughput with different numbers of concurrency is negligible on average and 3% maximal. The experiments show that Cruiser is feasible to be applied.

Acknowledgments

The authors would like to thank Ori Shalev and Nir Shavit for sharing the non-blocking hash table code [62], Maged M. Michael for pointing to us the open source project "Amino Concurrent Building Blocks" [46], Andrew Appel and Xi Xiong for their valuable comments, and 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.

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] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[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] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10 (56), May 2000.

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

[10] CERT Advisory, CA-2001-19 CodeRed worm.

[11] CERT Advisory, CA-2002-33 Heap Overflow Vulnerability in Microsoft Data Access Components.

[12] CERT Advisory, CA-2003-20 SQLSlammer worm.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security '05*, pages 177–192, 2005.

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

[15] M. Conover. w00w00 on heap overflows, 1999. www.w00w00.org/files/articles/heaptut.txt.

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

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

[18] 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.

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

[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] T. Durden. Bypassing PaX ASLR protection. *Phrack*, 2002.

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

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

[24] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08*, pages 43–52.

[25] N. Hardy. The confused deputy. *ACM Oper. Syst. Rev.*, 22(4):36–38.

[26] T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *DISC '01*, pages 300–314.

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

[28] M. Herlihy. A methodology for implementing highly concurrent data structures. In *PPoPP '90*, pages 197–206.

[29] IBM. ProPolice detector. www.trl.ibm.com/projects/security/ssp/.

[30] IBM System/370 Extended Architecture, Principles of Operations. IBM Publication No. SA22-7085, 1983.

[31] 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.

[32] 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.

[33] M. Kaempf. Vudo malloc tricks. *Phrack*, 11(57), 2001.

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

[35] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, 1977.

[36] D. Lea. dlmalloc. http://g.oswego.edu/.

[37] P. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *IPDPS '10*, pages 1–12.

[38] R. Lemos. Counting the cost of Slammer, 2003. http://news.cnet.com/Counting-the-cost-of-Slammer/2100-1002_3-982955.html.

[39] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[40] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02*, pages 73–82.

[41] MSDN. Structured exception handling. http://msdn.microsoft.com/en-us/library/ms680657(VS.85).aspx.

[42] 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.

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

[44] NIST. SAMATE Reference Dataset. http://samate.nist.gov/SRD.

[45] G. Novark and E. D. Berger. Dieharder: securing the heap. In *CCS '10*, pages 573–584.

[46] Open Source project. Amino concurrent building blocks. http://amino-cbbs.sourceforge.net/.

[47] Open Source Project. libsigsegv. http://libsigsegv.sourceforge.net/.

[48] S. Prakash, Y.-H. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput.*, 43(5):548–559, 1994.

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

[50] G. Richarte. Four different tricks to bypass StackShield and StackGuard protection. Tech. report, Core Security Tech., 2002.

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

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

[53] 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.

[54] SecuriTeam. http://www.securiteam.com/.

[55] SecurityFocus. CVS directory request double free heap corruption, 2003.

[56] SecurityFocus. Mozilla Firefox and Seamonkey regular expression parsing heap buffer overflow, 2009.

[57] SecurityFocus. Wu-ftpd file globbing heap corruption, 2001.

[58] SecurityFocus. libHX 'HX_split()' remote heap-based buffer overflow, 2010.

[59] SecurityFocus. Lynx browser 'convert_to_idna()' function remote heap based buffer overflow, 2010.

[60] SecurityFocus. http://www.securityfocus.com/.

[61] SecurityFocus. Sudo password prompt heap overflow, 2002.

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

[63] Solar Designer. Non-executable user stack, 1997. http://www.openwall.com/linux/.

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

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

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

[67] US-CERT. Vulnerability notes database. www.kb.cert.org/vuls.

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

[69] 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.

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

[71] M. Zhivich, T. Leek, and R. Lippmann. Dynamic buffer overflow detection. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.