LETTER

iCruiser: An Improved Approach for Concurrent Heap Buffer Overflow Monitoring*

Donghai TIAN[†,††a)], ***Nonmember***, **Xuanya LI**[†††], ***Member***, **Mo CHEN**[†], ***and*** **Changzhen HU**[†], ***Nonmembers***

SUMMARY Heap buffer overflow has been extensively studied for many years, but it remains a severe threat to software security. Previous solutions suffer from limitations in that: 1) Some methods need to modify the target programs; 2) Most methods could impose considerable performance overhead. In this paper, we present `iCruiser`, an efficient heap buffer overflow monitoring system that uses the multi-core technology. Our system is compatible with existing programs, and it can detect the heap buffer overflows concurrently. Compared with the latest heap protection systems, our approach can achieves stronger security guarantees. Experiments show that iCruiser can detect heap buffer overflow attacks effectively with a little performance overhead.

key words: *heap buffer overflow, multi-core technology*

1. Introduction

Although buffer overflow problems have been well studied for over two decades, there are still many related vulnerabilities [11]–[13] that are being both discovered and exploited. According to the National Vulnerability Database [2], more than 300 buffer overflow vulnerabilities were reported in 2012. Once attackers exploit these vulnerabilities, they may change the target program's execution or even control the victim's entire system.

The main reason for buffer overflows is that the target program fails to check the buffer boundary when writing data to its buffer. In general, there are two categories of buffer overflows: stack-based and heap-based buffer overflows. In this paper, we focus on heap-based buffer overflow detection.

Previous solutions to detect heap buffer overflows have the following limitations: 1) Most methods impose high performance penalties [1], [7], [9], [10]. 2) Some methods require recompiling the source code of the target program [7], [9]. 3) Some methods only protect some specific libc functions [10]. 4) Some methods rely on special hardware [8].

Manuscript received August 21, 2013.
Manuscript revised November 20, 2013.

[†]The authors are with Beijing Institute of Technology, China.
[††]The author is with State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China.
[†††]The author is with National Engineering Laboratory for Information Security Technologies, Institute of Information Engineering, Chinese Academy of Sciences, China.

*This work is supported partially by the National High-Tech Research Development Program of China under Grant No. 2009AA01Z433 and the Open Foundation of State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences) under Grant No. 2013-4-1.

a) E-mail: donghaitad@gmail.com

DOI: 10.1587/transinf.E97.D.601

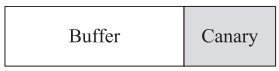

Fig. 1 Buffer structure.

Recently, Qiang et al. proposed a concurrent heap monitoring framework, Cruiser [5], which enables detecting heap buffer overflows efficiently without changing the target program. Although this method has made great progress on heap overflow detection, it still has some limitations: 1) The protected program and the associated monitor co-locate in the same address space so that the attacker could disable the monitor once the target program's execution is hijacked via a heap buffer overflow. 2) Since Cruiser simply relies on XOR operations to generate canaries to guard heap buffers (see Fig. 1), it is possible for attackers to recomputed and recover the corrupted canaries. To address these problems, Nick et al. presented a kernel-assisted heap protection system [1], which checks the canaries during the kernel's execution. Unfortunately, this system incurs considerable performance degradation.

In this paper, we present an improved heap protection system, `iCruiser`, which can detect heap buffer overflows concurrently. iCruiser is implemented on the Linux system, and it does not need any modification to the existing programs and OS kernel. Our approach makes the following contributions:

- We propose an improved heap overflow detection approach. This method can provide stronger security guarantees than current cutting edge heap protection mechanisms.
- We leverage the secure canary generation, inter-process communication mechanism and multi-core technology to achieve concurrent and secure monitoring heap buffer overflows.
- We design and implement a prototype of iCruiser based on Linux. Evaluations show that our system can detect heap buffer overflow efficiently.

2. Overview of Our Approach

The goal of iCruiser is to build a system that achieves concurrent heap buffer overflow monitoring with stronger security. Our high-level idea is consistent with previous canary-based methods: just to add one canary word to the end of each buffer, which is shown in Fig. 1. Whenever a heap

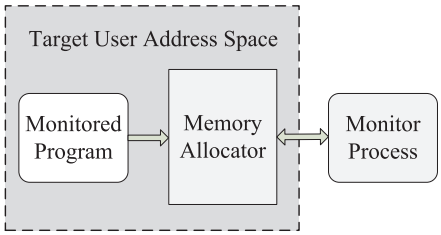

Fig. 2 System architecture.

buffer is overflowed, the canary value will be corrupted. Different from the conventional methods, we utilize a a cryptography method to generate secure canaries for different buffers. By doing so, it is difficult for attackers to infer and counterfeit the target canary value based on the other canaries.

Once the canaries are set, the next step is to check the integrity of these canaries. Traditional methods perform the checks during the heap allocate and de-allocate operations, which incur not only the performance overhead but also the detection latency. For example, when the program inline code performs the check, all the program's other jobs will be forced to wait. On the other hand, the heap overflows will not be detected until the allocation and de-allocation functions are invoked.

To overcome these problems, we propose to move the detection code out of the target program by utilizing the multi-core technology. In contrast to Cruiser [5] that places the monitoring code in the same protected process, we create a separate trusted process to monitor the protected program by utilizing the IPC shared memory. Thanks to the process isolation provided by the underlying OS, even if the protected process is compromised, the attacker cannot touch our monitor process. To ensure the protected program allocating dynamic buffers from the shared memory region that can be accessed by our monitor, we need to modify the default memory allocator. Similar to Cruiser, we utilize an express data structure to collect memory allocation information. Based on this data structure, our monitor can swiftly locate and check canaries.

The general architecture of our system is shown in Fig. 2. The monitor runs in a different process from the monitored program for self-protection. Before the monitored program gets executed, we load our memory allocator first instead of the original one. By doing so, our allocator can intercept the memory allocate and de-allocate operations issued by the monitored program. Moreover, the allocator will map the heap memory area of the monitored program to our monitor's address space so that our monitor can access the monitored program's heap for concurrent monitoring.

3. System Design and Implementation

We have developed iCruiser, a prototype based on Linux system to demonstrate our approach. We present our memory allocator in Sect. 3.1. The concurrent monitoring approach is described in Sect. 3.2.

3.1 Memory Allocator

Our allocator was implemented by modifying `dlmalloc` 2.8.3. The main function of our allocator is to hook the malloc calls and then attach secure canaries to the end of allocated buffers. To generate different canaries for different buffers, a conventional method may first invoke the srand() function to set a seed and then use the rand() function to get a random integer as the canary value. However, frequent invoking the rand() function would impose considerable performance cost. More importantly, the rand() function typically returns a 16-bit number, which is cyclic and predictable. As a result, attackers may figure out the canary value.

To address these problems, we apply a cryptography method for generating secure canaries. Specifically, we first utilize a kernel module to extract a random number from the entropy pool, which is maintained by Linux kernel. Then, the random number is used as a secret key for RC4 [4] to generate a rand stream of bytes. Finally, each 4 bytes of this stream is used as a canary value for each allocated buffer. To reduce the cost of canary generation, we generate a batch of secure canaries and store them in an array of the monitored program during its initialization. By doing so, we can simply obtain the canary from the array instead of regenerating it by using the kernel module. Since each canary in the array can be used only one time, we need to refill the array when all the generated canaries have been used. To this end, our allocator maintains a position pointer that points to the array entry, which stores a canary for the current use. Whenever the canary gets used, the pointer will be moved to the next canary position. Once the pointer is moved to the end of the array, our allocator will invoke the ioctl() system call to inform the underlying kernel module to regenerate secure canaries and then copy them into the user array.

To facilitate our monitor to access the monitored program's heap for checking canaries, we make use of the IPC mechanism to share the heap memory area with our monitor process. Specifically, our allocator first invokes the shmget() function to create an IPC shared memory region, whose default size is 512 MB. Then, it invokes the shmat() function to attach this shared memory region to its own address space. On the other hand, we apply the same functions shmget() and shmat() to get and then attach the same shared memory region to the monitor process. To ensure the allocated buffers only reside in the shared memory region, our allocator will only allocate memory in the shared memory segment but not in the traditional data segment. In addition, we need to deal with a special case when a very large chunk is requested. In this case, we may not find a proper chunk with large size in the shared memory region. As a result, we need to create another IPC shared memory region for this allocation.

To notify our monitor further locate and check the specific canaries, we need to transfer the canary location and value information to the monitor process. To this end, we

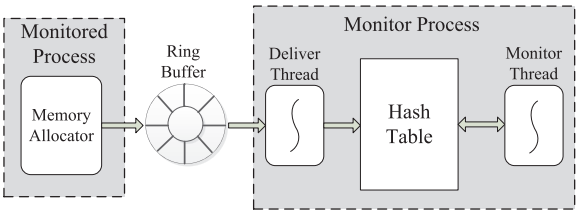

Fig. 3 Concurrent monitoring mechanism.

utilize the Lamport's ring buffer algorithm [6]. Since this algorithm allows the single-producer (i.e., the monitored program) and the single-consumer (i.e., our monitor) to access the ring buffer concurrently, the canary information in the buffer can be transferred efficiently.

When a buffer is freed, the corresponding canary information transferred to the monitor becomes dated and should be deleted. However, if our memory allocator releases this buffer directly without notifying the monitor, it may result in false alarms in that the dated buffer may be reused while the monitor is checking this buffer. To deal with this problem, our allocator just mark the target buffer with a tombstone flag and transfer this information to the monitor for later releasing the buffer.

3.2 Concurrent Monitoring

Once the canaries are set, our monitor is responsible for accessing the monitored program's heap and then check the canary. In order to improve our monitor's performance, we create two separate threads (i.e., deliver thread and monitor thread) in our monitor process, which is shown in Fig. 3. The deliver thread is used to copy the canary information from the ring buffer to a hash table, by which the monitor thread can quickly locate the canaries and then verify their integrity.

Since the deliver thread and monitor thread may access the hash table concurrently, we need to tackle the synchronization problem. To this end, a conventional method is to use a lock-based method. However, this method will degrade the performance. To avoid using locks, we turn to applying the lock-free hash table algorithm proposed by Shalev and Shavit [6]. Although doing so can achieve good scalability, the contention (i.e., insert and delete operations on the same node) may still lead to considerable performance overhead. Instead of relying on the existing methods, we have designed a custom lock-free hash table algorithm, which is shown in Fig. 4. Our algorithm supports a single deliver thread to insert nodes into a hash table and a single monitor thread to traverse this hash table and then delete the nodes concurrently.

Similar to the traditional hash table, we resolve the hash collision problem by using separate chaining. To deal with the contention problems, we separate the insert and delete operations in different areas. Specifically, when the hash table is initialized, we introduce a empty node for each hash table entry (Line 21~25). Then, we enforce each new node will be inserted into the hash table following the first empty node (Line 37~40).

When a buffer is freed, the corresponding node in the hash table should be removed because the related canary information becomes out of date. To delete the dated node,

```
1 #define HASH_SZ 4096
2 #define hash_fun(x) (((x)>>12)^(x))&(HASH_SZ-1)
3
4 struct canary_info{
5   unsigned long addr;
6   unsigned int size;
7   unsigned long value;
8 };
9
10 struct hash_item{
11   struct canary_info ca;
12   struct hash_item *next;
13 };
14
15 struct hash_item hash_tb[HASH_SZ];
16
17 void Insert_hashtable(struct hash_item *p)
18 {
19   int entry = hash_fun((p->ca).addr);
20   struct hash_item *htable;
21   if(hash_tb is not initialized){
22     for(int i=0; i<HASH_SZ; i++){
23       memset(hash_tb,0,sizeof(struct hash_item)*
             HASH_SZ); //Initialize empty nodes
24     }
25   }
26   htable = &hash_tb[entry];
27   if((p->ca).size==0){ //Update a node size
28     struct hash_item *cur = htable->next;
29     while(cur!=NULL){
30       if((cur->ca).addr==(p->ca).addr){
31         (cur->ca).size = 0;
32         break;
33       }
34       cur = cur->next;
35     }
36   }
37   else{ //Insert a node
38     p->next = htable->next;
39     htable->next = p;
40   }
41 }
42
43 void Monitor()
44 {
45   for(int i=0; i<HASH_SZ; i++){
46     struct hash_item *htable = &hash_tb[i];
47     struct hash_item *p = htable->next;
48     int count = 0;
49     while(p!=NULL){
50       Check_hash_item(p);//Check a buffer canary
51       if((p->ca).size==0 && count!=0){
52         Release((p->ca).addr);
53         Delete_hash_item(p);
54       }
55       p = p->next;
56       count++;
57     }
58   }
59 }
```

Fig. 4 Concurrent monitoring algorithm.

the deliver thread can perform this task directly. Instead of relying on the deliver thread, we utilize the monitor thread to carry out this deletion. In this way, the deliver thread can transfer the canary information efficiently. To mark the dated node for later deletion, we set the size field of the node as zero (Line 27∼36). Then, the monitor thread can check the node size to determine whether this node should be removed. Since the insert operations are always carried out following the first empty node, the deletion operations to be performed on the second node may result in contention problems. To tackle this issue, the marked node is delayed to get removed when it is no longer the second node (Line 51∼54).

It is worth noting that our monitor should release the corresponding buffer before deleting the dated node (Line 52∼53). For this purpose, the monitor should follow the allocator's rule to manipulate the metadata that is stored before the target buffer. Specifically, if the target buffer is not very large, we just mark the associated chunk that contains this buffer as available and then place it in the corresponding bin[†]. In particular, if the adjacent chunk is also free after the target chunk is released, then we coalesce these two adjacent chunks into one larger free chunk. Different from the original buffer release operations performed by `dlmalloc` allocator, we do not reclaim the allocated memory pages when the top contiguous chunks become free. The benefit is that the additional complex operations can be avoided. On the other hand, if the target buffer is within a very large chunk that is allocated separately via the IPC shared memory, our monitor will destroy this memory region.

4. Evaluation

In this section, we evaluate both the detection effectiveness and the performance of iCruiser. All the experiments are carried out on a Dell PowerEdge T410 work station with two 2.13G IntelXeon E5606 CPUs and 4 GB memory.

4.1 Effectiveness

We evaluate the effectiveness of iCruiser for heap buffer overflow detection with the SAMATE Reference Dataset (SRD) [3]. Particularly, we select 12 test cases on heap buffer overflows from this dataset. The experiments show that all these heap overflows are successfully detected by our system. Moreover, we exploit two recent heap buffer overflow vulnerabilities (i.e., libHX [11] and Lynx [12]). Specifically, these two vulnerable programs are first monitored by iCruiser, and then we utilize the corresponding heap overflow exploits to launch attacks. Our system detects these two overflow attacks by identifying the corrupted canaries.

[†]The `dlmalloc` allocator divides the heap memory into contiguous chunks with various size. The free chunks with the same size are kept in a doubly linked list, which is called a bin.

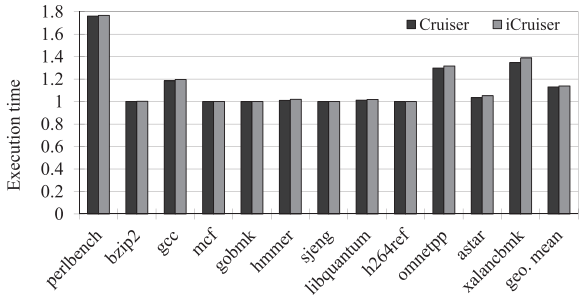

Fig. 5 SPEC CPU2006 performance (normalized to the execution time of native Linux).

4.2 Performance

To evaluate the performance of our system, we execute the SPEC CPU 2006 Integer benchmark suite, which is shown in Fig. 5. Compared with the original program, our system introduces a little performance overhead. In specific, the average performance overhead is 13.8%. In addition, we test the performance of the recent work, Cruiser. Compared with this system, the additional overhead added by our protection system can be negligible.

5. Related Work

Compared with our previous work Kruiser [14], there are three major differences in this work: (1) The protection targets are very different. Kruiser focuses on kernel-level heap buffer overflow monitoring, while our work aims at user-level heap buffer overflow detection. (2) The concurrent monitoring algorithms are quite different. Kruiser does not fully solve the synchronization problem, but it makes use of the double-check scheme to detect the unsynchronized states. On the contrary, we design a custom lock-free hash table algorithm to eliminate the synchronization issue in this work. (3) Kruiser requires some modification to the OS kernel and depends on the virtualization technology to achieve kernel heap buffer monitoring, while our approach does not need to modify the OS kernel and leverages the existing OS facilities to detect user heap buffer overflows.

6. Conclusion

In this paper, we present `iCruiser`, an improved heap buffer overflow detection system. We exploit secure canary generation, IPC shared memory and concurrent monitoring algorithm, which allows us to monitor heap buffer overflows efficiently with stronger security guarantees. Moreover, our protection system does not require modifying the target programs, so it can be deployed easily. Our evaluations show that iCruiser can detect heap buffer overflows effectively with good performance.

References

[1] N. Nikiforakis, F. Piessens, and W. Joosen, "HeapSentry: Kernel-

assisted protection against heap overflows," 10th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2013.
[2] NIST. National Vulnerability Database. http://nvd.nist.gov/, 2012.
[3] NIST. SAMATE Reference Dataset. http://samate.nist.gov/SRD, 2012.
[4] Wikipedia. RC4. http://en.wikipedia.org/wiki/RC4, 2012.
[5] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," Proc. 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI), 2011.
[6] L. Lamport, "Proving the correctness of multiprocess programs," IEEE Trans. Softw. Eng., vol.SE-3, pp.125–143, 1977.
[7] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," 21st International Conference on Distributed Computing Systems (ICDCS), 2001.
[8] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008.
[9] M. Zhivich, T. Leek, and R. Lippmann, "Dynamic buffer overflow detection," Proc. 11th Annual Network and Distributed System Security Symposium (NDSS), 2004.
[10] E.D. Berger, "HeapShield: Library-based heap overflow protection for free," University of Massachusetts Amherst, Tech Report, 2006.
[11] SecurityFocus, libHX 'HX split()' remote heap-based buffer overflow, 2010.
[12] SecurityFocus, Lynx browser 'convert to idna()' function remote heap based buffer overflow, 2010.
[13] SecurityFocus, Mozilla Firefox and Seamonkey regular expression parsing heap buffer overflow, 2009.
[14] D. Tian, Q. Zeng, D. Wu, P. Liu, and C. Hu, "Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring," Proc. 19th Annual Network and Distributed System Security Symposium (NDSS), 2012.