PAPER *Special Section on Security, Privacy and Anonymity in Computation, Communication and Storage Systems*

Mitigating Use-After-Free Attacks Using Memory-Reuse-Prohibited Library*

Toshihiro YAMAUCHI[†a)], ***Member***, **Yuta IKEGAMI**[†], ***and*** **Yuya BAN**[†], ***Nonmembers***

SUMMARY Recently, there has been an increase in use-after-free (UAF) vulnerabilities, which are exploited using a dangling pointer that refers to a freed memory. In particular, large-scale programs such as browsers often include many dangling pointers, and UAF vulnerabilities are frequently exploited by drive-by download attacks. Various methods to prevent UAF attacks have been proposed. However, only a few methods can effectively prevent UAF attacks during runtime with low overhead. In this paper, we propose HeapRevolver, which is a novel UAF attack-prevention method that delays and randomizes the timing of release of freed memory area by using a memory-reuse-prohibited library, which prohibits a freed memory area from being reused for a certain period. The first condition for reuse is that the total size of the freed memory area is beyond the designated size. The threshold for the conditions of reuse of the freed memory area can be randomized by HeapRevolver. Furthermore, we add a second condition for reuse in which the freed memory area is merged with an adjacent freed memory area before release. Furthermore, HeapRevolver can be applied without modifying the target programs. In this paper, we describe the design and implementation of HeapRevolver in Linux and Windows, and report its evaluation results. The results show that HeapRevolver can prevent attacks that exploit existing UAF vulnerabilities. In addition, the overhead is small.

key words: *use-after-free (UAF) vulnerabilities, UAF attack-prevention, memory-reuse-prohibited library, system security*

1. Introduction

Recently, there has been an increase in use-after-free (UAF) vulnerabilities, which can be exploited by referring a dangling pointer to a freed memory. A UAF attack abuses the dangling pointer that refers to a freed memory area and executes an arbitrary code by reusing the freed memory area. The number of UAF vulnerabilities based on the investigation in [2] is shown in Fig. 1. The figure shows that the number of UAF vulnerabilities has rapidly increased since 2010 [2]. Furthermore, the number of exploited UAF vulnerabilities has increased in Microsoft products [3]. In particular, large-scale programs such as browsers often include many dangling pointers, and the UAF vulnerabilities are frequently exploited by drive-by download attacks. For example, many UAF attacks exploit the vulnerabilities of plug-ins (e.g. Flash Player) in browsers. As a modern browser has a JavaScript engine, an attacker can exploit the UAF vulnerabilities using JavaScript, which creates and frees memory area.

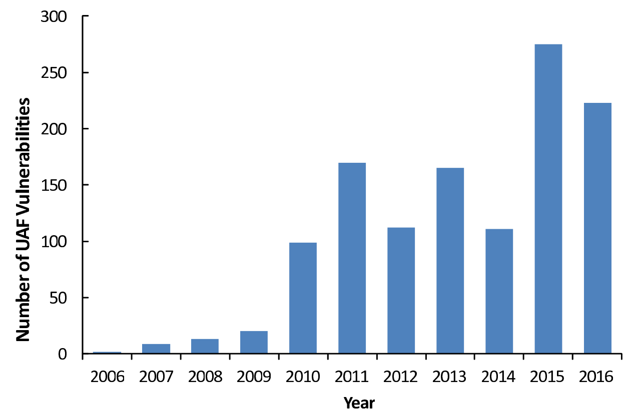

Fig. 1 Number of UAF vulnerabilities registered to CVE.

We investigated the kind of applications that include UAF vulnerabilities from the UAF vulnerabilities occuring in 2016 from the CVE database. The number of UAF vulnerabilities in 2016 was 224. We classified the types of applications. The investigation results showed that approximately 46% of the UAF vulnerabilities were included in browsers and browser plugins. These results also implied that browsers and their plugins included many UAF vulnerabilities. Therefore, countermeasures against UAF attacks are required for these applications.

To show the characteristics of a UAF attack, we investigated CVE-2012-4792, CVE-2012-4969, CVE-2013-3893, and CVE-2014-1776 as UAF vulnerabilities used for attacks in the real world. Investigation results demonstrated that in a UAF attack, the memory is immediately reused after a target freed-object is reused to reduce the possibility of a target memory area being reused by another process after it is released. This is because memory allocator deploys best fit algorithm. In this case, the chunk is immediately reused if the size of the chunk is the same as the requested size. The memory allocator maintains locality for minimizing page faults and cache misses. Maintaining the locality improves the performance on modern processors. As a result, recently freed chunks are immediately reused in such a memory allocator. Therefore, the operations of the current memory allocator are deterministic, and the attacker can guess the operations and reuse the target freed-memory. Various methods to prevent UAF attacks have been proposed [4]–[21]. However, only a few methods can effectively prevent UAF attacks during runtime with a low overhead. Furthermore, the

Manuscript received January 15, 2017.
Manuscript revised May 24, 2017.
Manuscript publicized July 21, 2017.
[†]The authors are with the Graduate School of Natural Science and Technology, Okayama University, Okayama-shi, 700–8530 Japan.
[*]A preliminary work of this paper was presented at [1].
a) E-mail: yamauchi@cs.okayama-u.ac.jp
DOI: 10.1587/transinf.2016INP0020

Copyright © 2017 The Institute of Electronics, Information and Communication Engineers

memory usage of the existing methods is inefficient, and these methods utilize a considerable memory area for preventing UAF-attacks.

Thus, many related works have used techniques, such as the DelayFree deploy technique that delays the time of freeing a memory object. References [22]–[24] also proposed methods to prevent UAF attacks against Internet Explorer (IE) by calling functions that have recently taken measures against UAF attacks. However, DelayFree [23] and Memory Protector [24] do not release the freed memory areas for a fixed period, thus complicating the UAF attacks. This period remains until the total size of the freed memory area is more than the threshold (beyond 100 KB). However, when the freed total memory size increases beyond the threshold, all the memory areas prevented to be released are released and can be reused. Each program must be altered to apply these methods, thereby resulting in the increase in the man-day requirement to modify a program and develop a patch. An attack against DelayFree is reported in [25], which indicated that an attack against DelayFree will succeed. An attack against IE secured using Isolated Heap and Memory Protector was also reported in [26]. Therefore, new countermeasures are required to prevent UAF attacks.

In this paper, we propose HeapRevolver, which is a novel UAF-attack prevention method that delays and randomizes the release timing of a freed memory area by using a memory-reuse-prohibited library. By delaying release of freed memory area, HeapRevolver prohibits the reuse of the memory area for a certain period. In UAF attacks, the freed memory area is immediately reused after the memory area is released to exploit the UAF vulnerabilities. Thus, the above-mentioned UAF attacks are prevented. The threshold for the conditions of reusing the freed memory area can be randomized by HeapRevolver. This function makes it more difficult to reuse the memory area for the UAF attacks by randomizing the timing of the memory area release. Accordingly, we added a reuse condition, in which the freed memory area is merged with an adjacent freed memory area before release. By adding this condition, a UAF attack will fail if an offset of the dangling pointer to the memory area is not appropriately calculated. Furthermore, HeapRevolver can be implemented in a library and be applied without altering the targeted program for protection. Thus, applying HeapRevolver to targeted programs is not difficult. As HeapRevolver can reuse the freed memory area under the reuse conditions, the memory can be efficiently used. Finally, we describe the design and the implementation of HeapRevolver in Linux and Windows. We report on the evaluation results that showed that the performance overhead of HeapRevolver is relatively smaller than that of DieHarder [21], which is one of the representative methods used to prevent UAF attacks by library replacement.

The differences between HeapRevolver and DelayFree are as follows:

1. The designated size for the conditions of reusing the freed memory area can be randomized by HeapRevolver. The designated size of DelayFree is constant.
2. A reuse condition in which the freed memory area is merged with an adjacent freed memory area before release, is newly introduced in HeapRevolver.
3. The released memory size is at most half of the designated size in the freed memory in HeapRevolver. In DelayFree, all the memory areas are released when a reuse condition is satisfied, and they can be reused.
4. HeapRevolver can be implemented in a library and applied without altering the targeted program for protection.
5. Shared libraries are often used in various OS's. Thus, HeapRevolver is applicable to various environments.

The preliminary version of this study appeared in [1]. We have added new survey and investigation results and additional evaluation results. The analysis has been provided in detail.

2. Problem and HeapRevolver Design

2.1 Problem of Existing Methods

The related studies [22]–[24] face the following problems:
Problem 1: the reuse timing can be guessed by attackers. The related methods do not release the freed memory area for a fixed period and complicate UAF attacks. The attackers can guess the reuse timing because of the period being fixed. Thus, the reuse time estimation must be made difficult.
Problem 2: need to alter the program code. Some methods alter the code of IE and call the recently added functions, thus preventing a UAF attack. Altering a program code is, therefore, necessary.
Problem 3: the target application and the OS's are limited. The methods protect IE in Windows against UAF attacks. Therefore, a more easy deployment method for various OS's and application programs is required for UAF attack mitigation.

In this paper, we propose a novel UAF attack-prevention method that will be used to resolve these three problems.

2.2 HeapRevolver Design

In this paper, we focus on the objective that the UAF attacks can be prevented by preventing the reuse of the freed memory area. However, when the reuse of freed memory area is prevented, memory usage becomes extremely inefficient. In addition, the overhead of creating a new memory area increases because *brk* and *sbrk* system calls are issued to expand the heap area. To solve this problem, we prohibit the reuse of a memory area for a certain period after it is freed. When a certain period has passed, the memory area can be reused. We assume that if this period is fixed, the reuse timing can be predicted by the attackers. Therefore, we randomize the prohibited period of the reuse in HeapRevolver.

To prevent UAF attacks reusing the memory objects, HeapRevolver alters an existing library. The altered library prohibits reuse of the freed memory for a certain period. The conditions for reuse are as follows:
Condition 1: The total size of the freed memory area is beyond the designated size.
Condition 2: The freed memory area is merged with an adjacent freed memory area.

When condition 1 is satisfied, the memory area that satisfies condition 2 is released. The released memory size is at most half of the designated total size in the freed memory. Condition 1 refers to technique used in DelayFree [23] and Memory Protector [24]. These techniques can prevent the immediate reuse of the freed memory area immediately after it is freed. However, the designated total size (threshold) in the freed memory in these techniques is constant. The threshold is 100 KB. When an attacker creates a memory area of 100 KB, the freed memory is released. Thus, an attacker can attempt to reuse a memory area by creating a memory area. Therefore, UAF attacks can be attempted.

In HeapRevolver, we develop two countermeasures for this problem. First, the total size threshold of the freed memory area is set to a larger value than that in DelayFree. This measure increases the threshold entropy against UAF attacks because threshold estimation becomes more difficult. Second, the threshold is randomized in some ranges. In addition, the threshold is randomly updated when condition 1 is satisfied. HeapRevolver releases at most only half of the freed memory area, implying that the randomly selected memory is delayed. These results in a certain memory area that cannot be reused for a long period. Therefore, UAF attacks become more difficult because the target memory object of an attacker cannot be reused for a long time. Furthermore, by adding condition 2, a UAF attack fails if an offset of a dangling pointer to the memory area is not appropriately calculated.

The C++ language is often used in large-scale programs as a web browser. The C++ program in Linux links libstdc++ library and is executed. The libstdc++ library includes the glibc library. Thus, a *new* operator and a *delete* operator finally call *malloc* and *free* functions in the glibc library. Therefore, HeapRevolver can be applied to the libstdc++ library by altering the glibc library and can protect programs based on the C++ language.

Three advantages can be gained by altering the existing library. First, shared libraries are often used in various OS's. Thus, this approach is applicable to various environments. Second, this approach does not need to modify existing programs. It simply needs to replace the existing library with a library applicable to our proposed method. Third, the cost of introducing HeapRevolver is reduced because it does not require program modification. In addition, HeapRevolver requires altering only the *free* function in Linux and the *HeapFree* function in Windows, and not any other function. Thus, the altering of functions is limited, and deploying and implementing HeapRevolver are easy.

3. Implementation of HeapRevolver

3.1 Implementation of HeapRevolver in Linux

In this section, we describe the implementation of HeapRevolver for glibc (x86_64) in Linux by altering only the *free()* function of the malloc algorithm that releases the memory area. Figure 2 shows the memory structure of malloc in HeapRevolver.

First, we explain the process of the original *free()* function of the malloc algorithm where the freed memory area is not created by the *mmap()* function. glibc manages the freed memory area in the *malloc_state* structure in the *malloc* library as a data structure called chunk. A freed chunk is inserted in lists called *fastbins* and *unsorted_chunks*. When the size of the freed chunk is below or equal to 128 bytes, it is inserted in *fastbins*. When the size of the freed chunk is bigger than 128 bytes, the freed memory area is merged with an unused adjacent memory area (if the adjacent memory area is unused) and inserted in *unsorted_chunks*. If not, the freed memory area is not merged and is inserted in *unsorted_chunks*.

In the *malloc()* function process, the lists are checked to find a new memory area. Then, an appropriate chunk is used.

Next, the *free()* function process of HeapRevolver is explicated as follows: *Lock_bins* and *wait_bins* are added to the *malloc_state* structure for HeapRevolver.

(1) The freed memory area (chunk) is stored in the head of the list (*lock_bins*).

(2) When the total size of the freed chunk stored in *lock_bins* and *wait_bins* is beyond the threshold limit, the freed chunks are released from the *lock_bins* list until half of the designated total size is released. The freed chunks must be merged with a *chunk* located in an adjacent memory cell before the chunks are released. When a freed chunk is removed from the *lock_bins*, HeapRevolver searches for a freed chunk that can be merged with the adjacent chunk from the *wait_bins* and *unsorted_chunks*. If HeapRevolver finds a chunk for merging, the freed chunk is merged with it

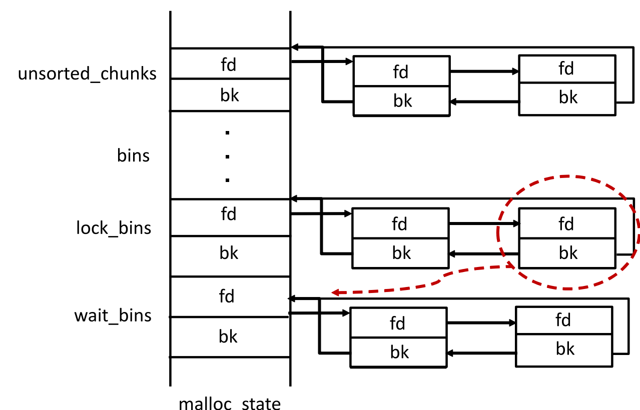

Fig. 2 Memory structure of malloc in HeapRevolver.

and is entered into the *unsorted_chunks* for release.

(3) If no chunk can be merged, the chunks in *lock_bins* are moved to *wait_bins* after attaching an attribute, indicating means that the chunk must be merged before reuse.

We believe that the threshold for the total size of the freed chunks is 1 MB, which is sufficient to complicate UAF attacks. In glibc of Linux/x86_64, a memory area that is larger or equal to 128 KB is created by the *mmap()* function. Thus, if the chunk size is smaller than 128 KB, the chunk is entered in the *lock_bins*. Therefore, more than seven chunks are entered in *lock_bins* when the threshold ≥ 1 MB. Furthermore, HeapRevolver randomizes the threshold of the total size when the total size of freed memory is larger than the threshold value. We assume that the threshold is randomized between some ranges designated by the users or administrators.

The proposed method is applied to a library, which is introduced by replacing an existing library in a specific directory or changing a linked dynamic library before it is loaded. For example, a linked dynamic library can be changed by modifying the path names of LD_PRELOAD and LD_LIBRARY_PATH.

3.2 Implementation of HeapRevolver in Windows

Windows' APIs *kernel32.dll* and *ntdll.dll* provide similar memory management processing as the glibc library in Linux. In addition, the *HeapFree()* function in *kernel32.dll* is often used to release a heap area. Thus, we implemented a function of HeapRevolver in the *HeapFree()* function. In our implementation, the *HeapFree()* function is hooked by our original function.

The hook function of HeapRevolver is implemented using a dynamic link library (DLL) injection and Windows API hook. DLL injection is a DLL mapping method to other processes and executes DLL processing in the processes. Windows API hook is a method that hooks a Windows API call and executes a certain processing before the hooked Windows API call. We deployed an import address table (IAT) hook for the Windows API hook. The address of the API functions exported from DLL is stored in IAT during the loading-process time. IAT hook is a method that modifies the address of APIs in IAT to call a target function.

Figure 3 shows the flow of hooking the *HeapFree()* function to the target process. First, HeapRevolver maps *Hook.dll* that performs IAT hook to a target process by DLL injection. Next, *Hook.dll* overwrites the address of the *HeapFree()* function stored in IAT in the address of the *Hook_HeapFree()* function of *Hook.dll*. When the *Hook_HeapFree()* function of *Hook.dll* is called by IAT hook, the *Hook_HeapFree()* function of *Hook.dll* obtains the arguments of the *HeapFree()* function and stores them in a ring buffer. Next, the *Hook_HeapFree())* function checks whether the sum of the freed memory are beyond the threshold. If the sum exceeds the threshold, the *Hook_HeapFree()* function obtains the arguments of the *HeapFree()* function and calls the *HeapFree()* function to release the freed memory area. The *Hook_HeapFree()* function calls the *HeapFree()* function until half of the threshold is released. If the sum of the freed memory does not exceed the threshold, the proposed function returns without any operation. Thus, the *Hook_HeapFree()* function delays the release of the freed memory area until the sum of the freed memory area exceeds the threshold.

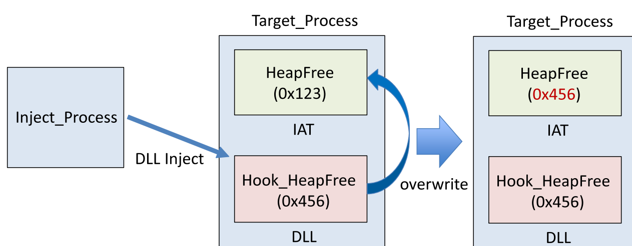

Fig. 3 Flow of hooking *HeapFree()* function on Windows.

The implementation of HeapRevolver in Windows is almost the same as in Linux. However, the prototype implementation of Windows does not include the determination of whether or not a memory area is already merged with an adjacent memory area. This point needs to be further studied. In addition, the prototype implementation in Windows uses the number of freed memory areas as a threshold instead of the sum of the freed memory area sizes because the process of managing the size is complex. Even when the amount of freed memory area is used as a threshold, the entropy can increase and complicate UAF attacks using a large threshold and randomizing it.

4. Evaluation

4.1 Security Analysis

4.1.1 Possibility of Success of UAF Attacks in HeapRevolver

We analyzed the possibility of attacks against HeapRevolver. For an attack to succeed, an attacker must reuse the freed memory area and overwrite the memory. Subsequently, malicious codes must be executed by referring to a dangling pointer. In HeapRevolver, the freed memory area cannot be reused until it satisfies the reuse condition because the area is entered into a wait_bin queue. Thus, most of the aforementioned UAF attacks can be prevented using HeapRevolver. Only when a memory area is freed, the sum of the freed memory area exceeds the threshold and the target memory area is merged to an adjacent memory area. The freed memory area can then be immediately reused after it is released. However, reusing the freed memory area is difficult in this case because the attacker must predict the size of the merged memory area (described in the next paragraph). In addition, the attacker must understand the number and the total size of the freed memory areas. The threshold of reuse is randomly set when the freed memory area is released and large-scale programs, such as browsers, process many mem-

ory allocations and releases. Hence, predicting when the sum of the freed memory area exceeds the threshold is very difficult.

The additional condition for attacks is the immediate reuse of the freed memory area after it is released. The requested size of memory allocation in many attacks is the same as that of the target freed memory area. In HeapRevolver, the reusable memory area must be merged to an adjacent memory area. Thus, the possibility of reuse is considerably reduced when the same size is designated for the memory allocation. For example, in Linux, unused memory area with a size is the same as the requested size is reused prior to the reuse of the memory area with another size.

If a dangling pointer is referred to before all the previous conditions are satisfied, the attacks will fail because of segmentation or other faults. After the faults, the application is terminated, and the next attack becomes impossible. Such failure in attacks reveals the attempts of attacks. Therefore, we believe that the attackers will avoid performing low-possibility attacks.

4.1.2 Attack Possibility against HeapRevolver

To defeat HeapRevolver, attackers consider repeating memory allocation and releasing memory. In addition, to increase the probability of successful attacks, heap spraying is used. Heap spraying is effective when the memory layout is predictable or the memory fragmentation in the heap area is suppressed. In HeapRevolver, freeing the memory area is randomly delayed, and memory fragmentation, such as external fragmentation, in the heap area frequently occurs. In this situation, a large area of heap spraying is often allocated in the last part of the heap area. We believe that the success of heap spraying is low. For the attacks against HeapRevolver to succeed, both UAF attacks and heap spraying must succeed; thus, the possibility of the success of two attacks is low, and the risk of revealing attack attempt is high because of failures.

As a typical attack, to overwrite a freed memory area referred by dangling pointer, the attacker attempts to allocate a large memory area after the target memory area is freed. Next, the attacker overwrites the entire target memory area. Overwriting a large memory area is expected to improve the possibility of a successful attack. This type of attack can succeed after the target memory area is freed and reused. As aforementioned, reusing the target memory area is difficult. In addition, the timing of freeing the target memory area is non-deterministic. Thus, creating attack codes with a high success probability against HeapRevolver is difficult.

4.2 Evaluation Environment

We used a computer with Intel Core i7-3770 (3.40 GHz) and 4-GB main memory for the evaluation. The OS's and versions used in the evaluations are Linux 3.13.0-45-generic/x86_64 (Ubuntu 14.04 LTS) and Windows 7 (64 bit). The HeapRevolver was implemented in glibc-2.19 in Linux.

To show the feasibility and overhead of the HeapRevolver, we evaluated the performance of HeapRevolver on Linux and Windows. The following experiments were performed: the UAF-attack-prevention experiments in Linux and Windows showed that UAF attacks can be prevented by HeapRevolver. In addition, we evaluated the performance overhead and memory usage of HeapRevolver. Finally, we compared HeapRevolver with DieHarder, which is one of the UAF prevention methods that use library replacement. In the overhead evaluations, we used fixed thresholds on HeapRevolver because we clarified the relationship between the threshold size and the performance and memory overhead of HeapRevolver.

We used four programs types for the evaluations. Browsers are targets of UAF attacks in the real world. We supposed that the main target for HeapRevolver protection is the browser programs. Thus, we used six browser benchmark programs [27]–[32] to evaluate the overhead.

Subsequently, HeapRevolver was implemented in the *free()* function in Linux and the *HeapFree()* function in Windows. We supposed that HeapRevolver affects the performance of memory allocation and release processes. Thus, we used the malloc-test program [33] as a memory-intensive program for the evaluations.

In addition, to evaluate the performance of the various processes, we used UnixBench [34], SysBench [35], and Himeno benchmark [36] for the evaluations.

4.3 Prevention Experiments of UAF Attack in Linux

We describe the experimental results of attempting UAF attacks using a program. In the program, an object of an *Addnum* class was created and deleted. Subsequently, when a memory area with the same size as that of the *Addnum* object was created, the memory area of the deleted *Addnum* object was reused. The address where a pointer of the shell code was stored was overwritten on the vtable address of the *Addnum* object. The shell code was executed by a call to the overwritten vtable. The program was executed when the address space layout randomization and data execution prevention were disabled.

Figure 4 shows the execution results before and after the application of HeapRevolver in Linux. Figure 4 (A) shows that the *Addnum* object and *buf* were allocated in the same memory area. Next, the UAF attack was performed by referring to a dangling pointer. Thus, the shell codes were executed. In contrast, Fig. 4 (B) shows that an *Addnum* object and *buf* were allocated in different memory areas. The memory area that did not include the pointer of the shell code was accessed by referring to the dangling pointer. Hence, the UAF attack failed due to the segmentation fault. HeapRevolver can prevent the UAF attack.

Table 1 Overheads in malloc-test.

Memory size	lib	thread num		
		1	3	5
100 B	glibc	0.335	1.02	1.71
	HeapRevolver (100 KB)	0.398 (18.8%)	1.200 (17.6%)	2.015 (18.1%)
	HeapRevolver (1 MB)	0.399 (19.1%)	1.205 (18.1%)	2.020 (18.4%)
512 B	glibc	0.371	1.132	1.885
	HeapRevolver (100 KB)	0.425 (14.5%)	1.310 (15.7%)	2.195 (16.4%)
	HeapRevolver (1 MB)	0.437 (17.8%)	1.324 (17.1%)	2.210 (17.2%)
1024 B	glibc	0.374	1.137	1.903
	HeapRevolver (100 KB)	0.526 (40.6%)	1.495 (31.5%)	2.481 (30.4%)
	HeapRevolver (1 MB)	0.543 (45.2%)	1.503 (36.6%)	2.509 (31.8%)

Table 2 Evaluation results on UnixBench, SysBench, and Himeno benchmark.

lib	UnixBench	SysBench (s)	Himeno benchmark
glibc	4,139.18	25.98	2,690.24
HeapRevolver (100 KB)	4,131.38 (0.19%)	26.21 (0.23%)	2,689.64 (0.02%)
HeapRevolver (1 MB)	4,130.57 (0.21%)	26.22 (0.24%)	2,688.05 (0.08%)

```
yuta@debian:~$ ./uaf 100 10
result = 110
Addnum = 0x602010
buf = 0x602010
$
```

(A) Before application of HeapRevolver

```
yuta@debian:~$ LD_PRELOAD="/usr/local/test2
/lib/libc.so.6" ./uaf 100 10
result = 110
Addnum = 0x602010
buf = 0x602030
Segmentation fault
```

(B) After application of HeapRevolver

Fig. 4 Experimental results of UAF attack prevention in Linux.

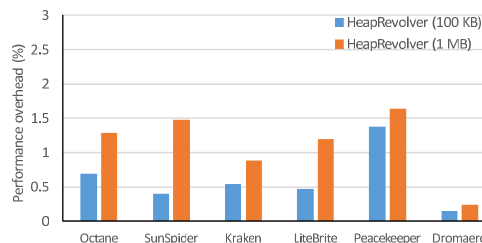

Fig. 5 Performance overhead of browser benchmarks on Firefox.

4.4 Evaluation of the Performance Overhead in Linux

To compare the performances of HeapRevolver and the original glibc, they were evaluated using several program types. The HeapRevolver thresholds in the evaluation were 100 KB and 1 MB.

First, the malloc-test benchmark was used to evaluate the processing time. The malloc-test benchmark contained some tests for the malloc and freeing processes. The tests were performed by multi-threading. The processing time was measured when the process was repeated for 10,000,000 times. The requested memory sizes were 100, 512, and 1,024 bytes. The number of threads was changed from one to five.

Table 1 lists the evaluation results, which showed that the overhead of HeapRevolver was less than 20% in the malloc-test when the memory sizes were 100 and 512 bytes. The HeapRevolver overhead increased by approximately 30%–45% when the requested memory size was 1024 bytes. We believe that this increase caused the repeated issue for the *sbrk* system call to change the size of the data segment in this evaluation. The evaluation results showed that the large threshold of the HeapRevolver involved a large overhead for every requested memory size.

The performance overhead of the HeapRevolver was then measured using the UnixBench, SysBench and Himeno benchmarks. Table 2 lists the evaluation results, which showed that the HeapRevolver overhead was less than 0.25% in every benchmark evaluation. The performance overhead of the 1-MB HeapRevolver was greater than that of the 100-KB HeapRevolver. We supposed that the performance overhead increased according to the size of the threshold, and that the performance overhead was small and acceptable.

Next, the overhead in applying the proposed method to glibc was measured using browser benchmarks. We used Firefox and Chrome as browsers for the evaluation. The processing time of the browser benchmarks was measured using Google's Octane 2.0, Apple's SunSpider 1.0.2, Mozilla's Kraken 1.1, Microsoft's LiteBrite, FutureMark's Peacekeeper, and Mozilla's Dromaeo. The measurement results were the average score of three execution times. Figures 5 and 6 show the comparison results of HeapRevolver with glibc in Firefox and Chrome, respectively, considering their performance overhead.

Figure 5 shows that the overhead was less than 1.8% in both 100 KB and 1 MB in Firefox. The overhead in the 1 MB HeapRevolver, in which the reuse duration was longer, was larger than that in the 100 KB HeapRevolver because the change in the amount of the data segment size (heap area), such as *sbrk* system call, increased when allocating a

Table 3 Response time (overheads) of thttpd web server (ms).

Method	Request file size			
	100 bytes	1 KB	10 KB	100 KB
glibc	74.0	75.3	131.1	1,057.8
HeapRevolver (100 KB)	77.1 (4.2 %)	80.0 (6.3%)	130.6 (−0.4%)	1,053.4 (−0.4 %)
HeapRevolver (1 MB)	77.6 (4.9 %)	76.4 (1.5 %)	131.4 (0.2 %)	1,057.9 (0.0 %)

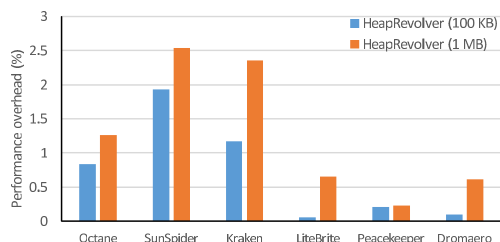

Fig. 6 Performance overhead of browser benchmarks on Chrome.

new memory area. Figure 6 illustrates that the overhead in Chrome was less than 2.6% in both the 100 KB and 1 MB HeapRevolvers. The overhead of the 1 MB HeapRevolver in Chrome was larger than that of 100 KB in Firefox.

Finally, the response time of a web server was measured. The thttpd 2.25b was used as a web server, while ApacheBench was used as a benchmark in measuring the response time of the web server in this evaluation. A total of 50 concurrent accesses were executed in this evaluation. Each access was repeated 1,000 times. The size of the requested file varied from 100 bytes, 1 KB, 10 KB, and 100 KB.

Table 3 lists the evaluation results of the response time of thttpd and shows that the overhead of HeapRevolver in every result was small. However, the overhead of HeapRevolver increased when the requested file size was 100 bytes. These process included network and CPU processes. Thus, we assume that the overhead of the memory allocation and release were hidden by network processes in cases where the size of the requested file is larger than 100 bytes.

The results of the abovementioned four evaluations showed that the HeapRevolver overhead was small. However, in the memory allocation and freeing processes, the results showed that the performance overhead affected the performance of the program.

4.5 Evaluation of Memory Consumption in Linux

We performed three experiments to evaluate the memory consumption of HeapRevolver in Linux. The thresholds of HeapRevolver were 100 KB and 1 MB.

We measured the memory usage of the malloc algorithm with HeapRevolver and compared it with that of original glibc. We used a malloc-test program. Five threads were run in this experiment, and the allocation and freeing processes were performed when the memory size was 512 bytes. Each thread repeated this process for 10 million times. We measured the memory usage when the processing of the five threads was finished.

Table 4 Memory usage of the malloc-test.

Method	Memory usage (KB)
glibc	588
HeapRevolver (100 KB)	588
HeapRevolver (1 MB)	1452

Table 5 Memory usage after Firefox finished browsing the 10 websites

Method	Memory usage (MB)
glibc	282
HeapRevolver (100 KB)	279
HeapRevolver (1 MB)	294

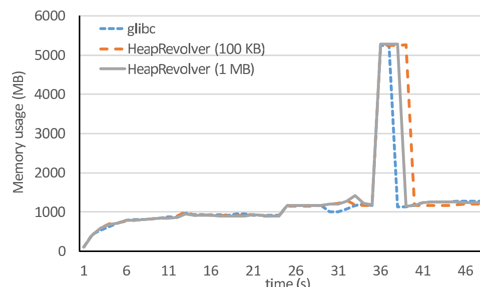

Fig. 7 Memory usage of Octane on Firefox.

Table 4 shows that the memory usages of glibc and the 100-KB HeapRevolver were almost the same. The size of the freed memory area was less than the threshold. When the threshold was 1 MB, the size of the exceeded memory usage was within the threshold limit. Therefore, these results implied that the maximum overhead of the memory usage for each process was less than the threshold.

We used Firefox 31.0 and Selenium IDE to evaluate the memory consumption when continuously browsing 10 websites. We then measured the memory consumption after Firefox finished browsing the 10 websites.

Table 5 lists the evaluation results of the website browsing. The memory usage of glibc and that of HeapRevolver were almost the same. The memory usage was between 279 and 294 MB because the memory usage overhead of HeapRevolver was small, and the memory usage variation was relatively large.

The change in the amount of virtual memory consumption when a browser benchmark was run was measured to compare HeapRevolver with glibc. Octane, SunSpider, and Kraken were used in this evaluation.

Figures 7–8 show the memory consumption of Octane in Firefox and Chrome. The evaluation results of Octane in Firefox and Chrome illustrated that the memory consumption of HeapRevolver was almost the same as that of glibc.

Table 6 Maximum memory consumption on browser benchmarks (KB).

Browser	lib	Octane	SunSpider	Kraken
Firefox	glibc	5,375,276	917,996	1,158,092
	HeapRevolver (100 KB)	5,382,988 (0.14 %)	922,416 (0.48 %)	1,124,996 (−2.86 %)
	HeapRevolver (1 MB)	5,407,820 (0.61 %)	949,344 (3.41 %)	1,151,620 (−0.56 %)
Chrome	glibc	1,441,932	1,431,016	1,421,312
	HeapRevolver (100 KB)	1,427,148 (−1.03 %)	1,414,824 (−1.13 %)	1,406,628 (−1.03 %)
	HeapRevolver (1 MB)	1,428,172 (−0.95 %)	1,415,848 (−1.06 %)	1,406,628 (−1.03 %)

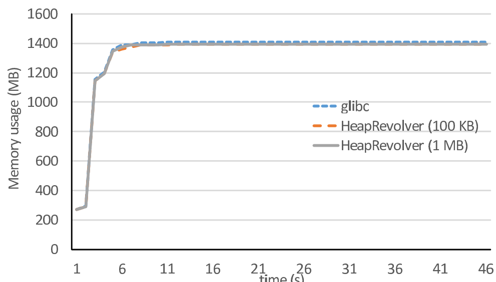

Fig. 8 Memory usage of Octane on Chrome.

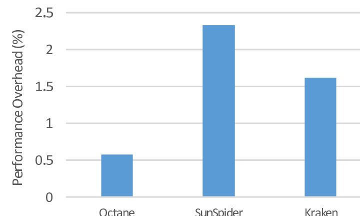

Fig. 9 Overheads of IE10 browser benchmark.

Furthermore, the memory consumptions of SunSpider and Kraken of the browser benchmarks in both browsers were almost the same as those of glibc. Therefore, the overhead in the memory consumption in HeapRevolver was also small.

Table 6 lists the maximum memory consumption under each condition. The maximum memory consumption of glibc of Octane in Firefox was 5,375,276 KB, while that in HeapRevolver of Octane in Firefox was 5,382,988 KB when the threshold was 100 KB and 5,407,820 KB when the threshold was 1 MB. The evaluation results showed the overhead of the maximum memory consumption in Firefox was small.

Table 6 indicates that the maximum memory consumption of glibc of Octane in Chrome was 1,441,932 KB, while that in HeapRevolver of Octane in Chrome was 1,427,148 KB when the threshold was 100 KB and 1,428,172 KB when the threshold was 1 MB. The evaluations of SunSpider and Kraken in Chrome demonstrated that the maximum memory consumption of HeapRevolver in Chrome was slightly smaller than that of glibc.

The amount of memory consumption with the introduction of HeapRevolver was small, as obtained from the abovementioned evaluation results.

4.6 Prevention Experiments against UAF Attack in Windows

We experimented on whether or not UAF attacks using real attack codes distributed in Metasploit [37] could be prevented. The attack codes used in the environments exploited CVE-2011-1260 and CVE-2012-4969 of IE 7 on Windows XP, CVE-2014-0322 of IE10, and CVE-2016-9079 of Firefox 38 on Windows 7. We determined that approximately 3,000 freed memory areas existed and were reserved for reuse in Linux when a threshold of 1 MB was set. Thus, we used 3,000 as the threshold for the Windows experiments.

We applied HeapRevolver to IE and Firefox on Windows as described earlier. The attack codes were then executed in each environment. Thus, HeapRevolver successfully prevented all the UAF attacks that reused memory objects. However, the UAF attacks sometimes succeeded when we applied HeapRevolver to IE and Firefox on Windows. We will describe the relation between the attack success rate and a threshold later.

4.7 Evaluation of the Performance Overhead in Windows

We measured the overhead of HeapRevolver both before and after the introduction of HeapRevolver on Windows 7. We ran three types of browser benchmark, namely, Octane, SunSpider, and Kraken, on IE 10. The HeapRevolver threshold was 3,000. Figure 9 shows the evaluation results.

The measured overhead of HeapRevolver in the three browser benchmarks was less than 2.5%. These browser benchmarks were CPU-intensive and required a large memory. Thus, we supposed that the influence on the performance of the browser benchmarks can be explicitly observed. Nevertheless, the results showed that the HeapRevolver overhead in Windows was small, and the overhead was acceptable.

4.8 Evaluation of Randomizing a Threshold

We first evaluated the relation between attack the success rate and a threshold to evaluate the effectiveness of randomizing a threshold. The attack codes used in the environments exploited CVE-2014-0322 of IE10, and CVE-2016-9079 of Firefox 38 on Windows 7.

Figure 10 shows the relation between the attack success rate and a threshold and illustrates that the attack success rate decreased according to the increase of a threshold. HeapRevolver were not applied to the browsers when the threshold is zero. The attack success rate of CVE-2014-0322 was equal to 10% when the threshold was equal to 4,000, and was zero when the threshold was equal to 5,000.

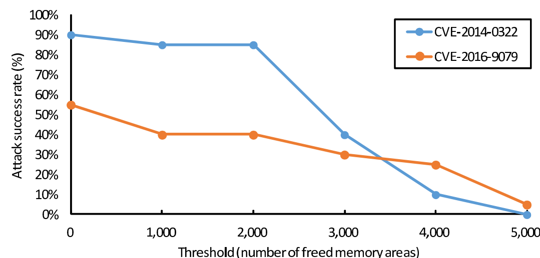

Fig. 10 Relation between the attack success rate and a threshold.

Table 7 Evaluation of randomized threshold (CVE-2014-0322).

Ranges of randomized threshold	Attack success rate
Without HeapRevolver	90%
From 3,000 to 5,000	20%
From 4,000 to 6,000	0%
From 5,000 to 7,000	0%

Table 8 Evaluation of randomized threshold (CVE-2016-9079).

Ranges of randomized threshold	Attack success rate
Without HeapRevolver	55%
From 3,000 to 5,000	40%
From 4,000 to 6,000	45%
From 5,000 to 7,000	35%
From 6,000 to 8,000	25%

The attack success rate of CVE-2016-9079 was equal to 25% when the threshold was equal to 4,000, and was 5% when the threshold was equal to 5,000. These results implied that HeapRevolver can effectively prevent the UAF attack against IE10 and Firefox. In addition, these results indicated that the HeapRevolver threshold should be randomized, where the threshold was equal to or more than 4,000. The relationship between the decrease in the attack success rate and the memory consumption was a tradeoff. Therefore, not only the attack success rate, but also the memory consumption amount must be considered.

Next, we evaluated HeapRevolver when the threshold was randomized in several cases to evaluate the effective randomized ranges of the threshold. Table 7 lists the evaluation results of the randomized threshold (CVE-2014-0322). The attack success rate in the ranges of 4,000 to 6,000 and 5,000 to 7,000 was zero. These results indicated that HeapRevolver can prevent UAF attacks when an appropriate threshold range is set.

Table 8 lists the evaluation results of the randomized threshold (CVE-2016-9079). The attack success rate in the four cases was from 25% to 45%. The attack success rate gradually decreased to the increase of a threshold. These results indicated that HeapRevolver can prevent UAF attacks. However, the attack success rate was not decreased sufficiently in the four cases. We will analyze the relation between attack success rate and ranges of randomized threshold in detail in the future.

The threshold was randomly chosen from the range each time the reuse condition was satisfied. We chose 2,000 as the width of randomly choosing the threshold. There-

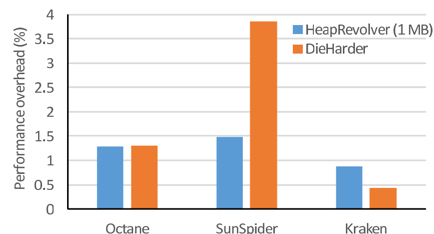

Fig. 11 Comparison of HeapRevolver and DieHarder for browser benchmarks in Firefox.

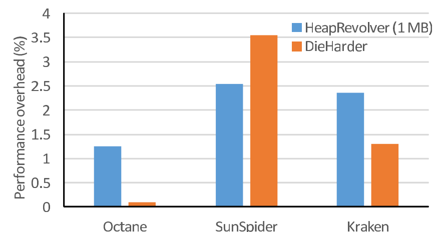

Fig. 12 Comparison of HeapRevolver and DieHarder for browser benchmarks in Chrome.

fore, we supposed that the entropy of the threshold was sufficiently high, and that it was sufficiently for an attacker to guess the threshold.

4.9 Comparison with the Existing Method

We compared HeapRevolver with DieHarder [21], which can be classified to be the same as HeapRevolver. The HeapRevolver threshold in this evaluation was 1 MB.

Figures 11 and 12 show the performance overhead of HeapRevolver compared with that of glibc when Octane, SunSpider, and Kraken were executed in Firefox and Chrome. The performance overhead of HeapRevolver was less than that of DieHarder, except in Kraken. The performance overhead of HeapRevolver was less than 3.0%, but the that of DieHarder in SunSpider was relatively large (i.e., approximately 4%). We will analyze the resultant factor of DieHarder in the future. However, we believe some inefficient processing in the reuse of objects in DieHarder occurred.

Table 9 lists the evaluation results of the malloc-test. The performance overhead of DieHarder was more than 200% that of glibc because DieHarder allocated memory area at random from some ranges in the memory area. The malloc-test is a memory-intensive program. Therefore, we supposed that the performance overhead increased compared with that in HeapRevolver.

In addition, we evaluated the performance overhead results of the original glibc using UnixBench, SysBench, and Himeno benchmarks (Table 10). The results showed that the performance overhead of HeapRevolver was smaller than that of DieHarder in all the benchmarks.

Finally, we evaluated the change in the amount of memory consumption under three browser benchmarks in Firefox. Figure 13 shows that the memory consumption of

Table 9 Evaluation results of malloc-test.

Memory size	lib	thread num				
		1	2	3	4	5
512B	HeapRevolver (1MB)	0.437 (17.8%)	0.880 (17.6%)	1.324 (17.1%)	1.765 (16.2%)	2.210 (17.2%)
	DieHarder	1.247 (236%)	2.586 (245%)	4.094 (262%)	5.421 (259%)	6.982 (270%)

Table 10 Evaluation results of UnixBench, SysBench, and Himeno benchmarks.

lib	UnixBench (KB/s)	SysBench (s)	Himeno benchmark
HeapRevolver (1MB)	4,130.57 (0.21%)	26.22 (0.24%)	2,688.05 (0.08%)
DieHarder	4,124.77 (0.35%)	26.25 (1.04%)	2,674.44 (0.60%)

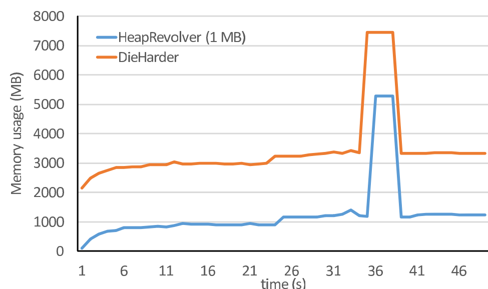

Fig. 13 Overheads of Firefox browser memory usage (Octane).

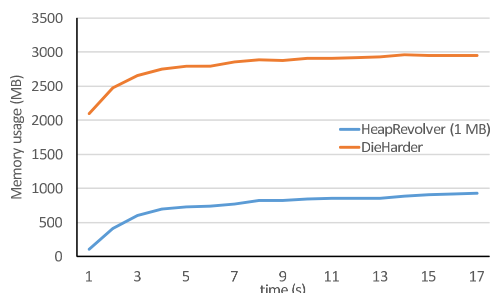

Fig. 14 Overheads of Firefox browser memory usage (SunSpider).

DieHarder in Octane was more than twice that of HeapRevolver. Figure 14 depicts that the memory consumption of DieHarder in SunSpider was approximately three times more than that of HeapRevolver. The overhead of DieHarder was very heavy to use in the real world. Comparatively, the results showed that the memory usage of HeapRevolver was efficient because HeapRevolver delayed the reuse of freed memory within the threshold size.

We now discuss the results in the Chrome browser. We evaluated the total memory consumption of the processes created by Chrome because Chrome creates more than one process. We measured the total memory consumption of virtual memory in all Chrome processes, and compared HeapRevolver with DieHarder. The total memory consumption of HeapRevolver in Octane was 45,904,020 KB, whereas that of DieHarder was 87,906,816 KB. These results implied that the memory consumption of DieHarder in Octane was approximately twice that of HeapRevolver. In addition, the memory usage trend in Chrome was similar to that in Firefox.

All comparison results demonstrated that the HeapRevolver overhead was smaller than that of DieHarder in most cases. Furthermore, the amount of memory consumption of HeapRevolver was less than that of DieHarder. In addition, source codes are necessary to apply DieHarder in Windows. The allocator must also be linked and compiled during the development process. In comparison, HeapRevolver does not need a source code and can be applied to programs, where the source codes cannot be obtained.

We compared HeapRevolver with DieHarder in the aspect of security. DieHarder is based on DieHard [38], [39] and randomizes the placement of allocated objects and the length of time before freed objects are recycled. The randomized placement can provide entropy for the position of allocated objects, and the entropy decreases the probability that overflow attacks will succeed. In contrast, HeapRevolver does not randomize the position of allocated objects. Thus, the probability of overflow attacks of HeapRevolver is higher than that of DieHarder. In addition, DieHarder introduces a sparse page layout, mapping a large fixed-size region of virtual address space, and sparsely using individual pages. They protect against heap spraying attacks by providing more entropy in object addresses. HeapRevolver does not deploy a sparse page layout and so on.

DieHarder randomly chooses newly allocated chunks across all the free chunks of proper size. DieHarder's overprovisioning ensures $O(N)$ free chunks. Therefore, the probability of returning the most recently freed chunk is low, but not zero. Meanwhile, HeapRevolver prohibits the reuse of the freed memory for a certain period, in which the probability of returning the most-recently freed chunk is zero. Thus, the probability of UAF attack reusing the most recently freed chunk of HeapRevolver is lower than that of DieHarder.

HeapRevolver focuses on the prevention of UAF attacks. Hence, the memory consumption of HeapRevolver does not significantly increase. In contrast, DieHarder prevents overflow attacks, heap spraying, and UAF attack. However, the overprovisioning of freed chunks of DieHarder causes the large memory overhead mentioned above. In addition, a sparse page layout increases the size of a process' page table, and sparsely using individual pages wastes physical memory.

5. Related Work

Dangling pointer-detection approaches [4]–[8] include dy-

namic binary translation, shadow memory, and taint analysis. These approaches detect dangling pointers before program execution. However, UAF attacks cannot be prevented in runtime if the dangling pointers are abused, which cannot be detected before a practical use. References [9] and [10] add codes that detect dangling pointers in a compilation and detect UAF attacks in runtime. References [9] and [11] detect dangling pointers by a static analysis of a binary program. Reference [12] inserts dynamic runtime checks for protecting against UAF vulnerabilities. This approach requires a source code.

UAF attacks in Refs. [13]–[15] were prevented by replacing a malloc library with a new library in which the allocation unit is a page. However, the memory usage is inefficient because the allocation unit of the created memory area consists of pages. Reference [16] proposes a library that reuses an object with the same size and alignment. This library writes data, whose type is different from that of the freed object, when UAF vulnerabilities are exploited. Therefore, a UAF attack that uses a different data type can be prevented. References [17] and [21] randomized the position of the created memory area. Some UAF attacks can be prevented using this approach.

In Refs. [18]–[20], a UAF attack was prevented using a method that prevents the alteration of vtable. Most UAF attacks rewrite the pointer stored in vtable, and an arbitrary code is executed. However, these methods cannot handle a UAF attack that does not alter vtable. In addition, rewriting the binary of a target program beforehand is required. The modification also depends on the binary form. Therefore, rewriting the AP binary is difficult.

Reference [40] presents a novel strategy to exploit the UAF vulnerabilities in Linux kernel and the mitigation schemes. The target is Linux kernel, but the target of HeapRevolver is the application programs in Linux and Windows. OpenBSD introduced the malloc function that uses the mmap function. The allocation of the memory areas by the malloc is randomized, and the memory area is unmapped from the process address space. The OpenBSD malloc function is based on the OpenBSD mmap system call. Thus, it cannot be applied to other OS's without the modification of OS kernels. In addition, the page cannot be unmapped until all the memory areas in a page are released. Thus, the unmapped memory areas that are released can be accessed by UAF attacks.

6. Conclusions

This study proposed HeapRevolver and described its design and implementation in Linux and Windows. The memory-reuse-prohibited library prevents the freed memory area from being reused during a certain period. Hence, the HeapRevolver can prevent UAF attacks without altering the targeted program for protection. The UAF attacks became more difficult because the timing of reuse of the freed memory area was randomized in HeapRevolver by randomizing the maximum total size of the freed memory areas (the HeapRevolver threshold).

The evaluation results in Linux showed that the HeapRevolver overhead was sufficiently small. However, the process of repeating memory allocation and releasing memory slightly influenced the performance. Furthermore, the evaluation results showed that the increase in the memory consumption was slight compared with that in the original glibc, and the overhead was acceptable. The experimental results in Windows using UAF exploit codes implied that UAF attacks can be prevented using HeapRevolver. The performance evaluation results by using browser benchmarks also showed that the HeapRevolver overhead was less than 2.5%. Finally, we compared HeapRevolver with DieHarder through evaluations. The results of the browser benchmarks demonstrated that the HeapRevolver overhead was smaller than that of DieHarder in most cases, and the amount of memory consumption of HeapRevolver was approximately half that of DieHarder.

Moreover, HeapRevolver can be easily deployed in existing systems and programs and can make UAF attacks more difficult. In addition, the HeapRevolver overhead was sufficiently small to be deployed in real systems. We believe that HeapRevolver can prevent UAF attacks by exploiting zero-day vulnerability.

Acknowledgements

We would like to thank Hiroyuki Uekawa of Okayama University for his support. This research was partially supported by Grant-in-Aid for Scientific Research 16H02829.

References

[1] T. Yamauchi and Y. Ikegami, "HeapRevolver: Delaying and randomizing timing of release of freed memory area to prevent use-after-free attacks," 10th International Conference on Network and System Security (NSS 2016), Lecture Notes in Computer Science (LNCS), vol.9955, pp.219–234 (2016). DOI: 10.1007/978-3-319-46298-1_15
[2] Common Vulnerabilities and Exposures, https://cve.mitre.org/index.html
[3] Microsoft Security Intelligence Report Volume 16, http://www.microsoft.com/en-us/download/details.aspx?id=42646
[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," 2012 USENIX Annual Technical Conference (USENIX ATC '12), pp.309–318, 2012.
[5] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early Detection of dangling pointers in use-after-free and double-free vulnerabilities," 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), pp.133–143, 2012.
[6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), pp.89–100, 2007.
[7] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp.213–223, 2011.
[8] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," 2015 Network and Distributed System Security Symposium (NDSS), 2015.

[9] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," 2014 Network and Distributed System Security Symposium (NDSS), 2014.
[10] C.F. Eigler, "Mudflap: Pointer use checking for C/C++," http://gcc.fyxm.net/summit/2003/mudflap.pdf
[11] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," Journal of Computer Virology and Hacking Techniques, vol.10, no.3, pp.211–217, 2014.
[12] Y. Younan, "FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers," 2015 Network and Distributed System Security Symposium (NDSS), 2015.
[13] GFlags and PageHeap, https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561%28v=vs.85%29.aspx
[14] Electric Fence, http://elinux.org/Electric_Fence
[15] D.U.M.A. - Detect Unintended Memory Access, http://duma.sourceforge.net/
[16] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," 19th USENIX Conference on Security (USENIX Security '10), pp.177–192, 2010.
[17] V.B. Lvin, G. Novark, E.D. Berger, and B.G. Zorn, "Archipelago: Trading address space for reliability and security," 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), pp.115–124, 2008.
[18] D. Dewey and T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code," 19th Network and Distributed System Security Symposium (NDSS), pp.1–14, 2012.
[19] C. Zhang, C. Song, K.Z. Chen, Z. Chen, and D. Song, "VTint: Protecting virtual function tables' integrity," 22nd Annual Network and Distributed System Security Symposium (NDSS), 2015.
[20] R. Gawlik and T. Holz, "Towards automated integrity protection of C++ Virtual function tables in binary programs," 30th Annual Computer Security Applications Conference (ACSAC '14), pp.396–405, 2014.
[21] G. Novark and E.D. Berger, "DieHarder: Securing the heap," 17th ACM Conference on Computer and Communications Security (CCS '10), pp.573–584, 2010.
[22] J. Tang, "Isolated heap for internet explorer helps mitigate UAF exploits," http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/
[23] J. Tang, "Mitigating UAF exploits with delay free for internet explorer," http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/
[24] Security Intelligence, "Understanding IE's new exploit mitigations: The memory protector and the isolated heap," https://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap/
[25] Security Week, "Microsoft's use-after-free mitigations can be bypassed: Researcher," http://www.securityweek.com/microsofts-use-after-free-mitigations-can-be-bypassed-researcher
[26] A.-A. Hariri, S. Zuckerbraun, and B. Gorenc, "Abusing silent mitigations - Understanding weaknesses within internet explorers isolated heap and memoryprotection," https://www.blackhat.com/us-15/briefings.html
[27] Octane 2.0, http://octane-benchmark.googlecode.com/svn/latest/index.html
[28] SunSpider 1.0.2 JavaScript Benchmark, https://www.webkit.org/perf/sunspider/sunspider.html
[29] Kraken JavaScript Benchmark (version 1.1), http://krakenbenchmark.mozilla.org/
[30] LiteBrite Benchmark, http://ie.microsoft.com/testdrive/Performance/LiteBrite/
[31] Peacekeeper, http://peacekeeper.futuremark.com/
[32] Dromaeo, http://dromaeo.com/
[33] C. Lever and D. Boreham, "Malloc() performance in a multithreaded Linux environment," Proc. USENIX Annual Technical Conference (ATC '00), pp.301–311, 2000.
[34] UnixBench, https://code.google.com/p/byte-unixbench/
[35] SysBench, https://launchpad.net/sysbench
[36] Himeno benchmark, http://accc.riken.jp/2444.htm
[37] Metasploit, http://www.metasploit.com/
[38] E.D. Berger and B.G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," Proc. 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06), pp.158–168, 2006.
[39] E.D. Berger and B.G. Zorn, "Efficient probabilistic memory safety," Technical Report UMCS TR-2007-17, Department of Computer Science, University of Massachusetts Amherst, 2007.
[40] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in Linux kernel," Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15), pp.414–425, 2015.

Toshiriro Yamauchi received B.E., M.E. and Ph.D. degrees in computer science from Kyushu University, Japan in 1998, 2000 and 2002, respectively. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate in Faculty of Information Science and Electrical Engineering at Kyushu University. He has been serving as associate professor of Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM, USENIX and IEEE.

Yuta Ikegami received B.E. and M.E. degrees in computer science from Okayama University, Japan in 2013 and 2015, respectively. His research interests include computer security.

Yuya Ban received B.E. degree in computer science from Okayama University, Japan in 2017. He entered Graduate School of Natural Science and Technology at Okayama University, 2017. His research interests include computer security.