HeapRevolver: Delaying and Randomizing Timing of Release of Freed Memory Area to Prevent Use-After-Free Attacks

Toshihiro Yamauchi and Yuta Ikegami

Graduate School of Natural Science and Technology, Okayama University,
3-1-1 Tsushima-naka, Kita-ku, Okayama, 700-8530 Japan

Abstract. 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. 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. 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.

Keywords: 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. Figure 1 shows the number of UAF vulnerabilities investigated in [1]. The figure shows that the number of UAF vulnerabilities has rapidly increased since 2010 [1]. Further, the number of exploited UAF vulnerabilities has increased in Microsoft products [2]. 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.

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

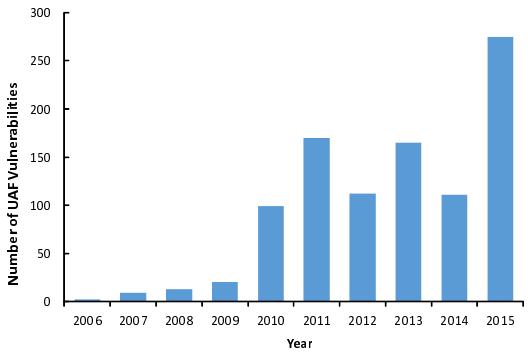

Fig. 1. Number of UAF vulnerabilities

used for attacks in real world. Investigation results show that in a UAF attack, memory is reused immediately 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. Various methods to prevent UAF attacks have been proposed [3]-[13]. However, only a few methods can effectively prevent UAF attacks during runtime with low overhead. Furthermore, the memory usage of existing methods is inefficient, and these methods utilize 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. In [15]-[17], methods were proposed to prevent UAF attacks against Internet Explorer (IE) by calling functions that have recently taken measures against UAF attacks. However, DelayFree [16] and Memory Protector [17] do not release the freed memory areas for a fixed period, thus complicating 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 memory areas that were prevented to be released are released and can be reused. In addition, each program must be altered to apply these methods, resulting in the increase in man-day requirement to modify a program and develop a patch. An attack against DelayFree is reported in [18], indicating that an attack against DelayFree will succeed. In addition, an attack against IE secured using Isolated Heap and Memory Protector was reported in [19]. 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. Thus, the abovementioned UAF attacks are prevented. The threshold for the conditions of reuse of the freed memory area can be randomized by HeapRevolver. This function makes it more difficult to reuse memory area for

UAF attacks by randomizing the timing of the release of the memory area. In addition, 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 implementation of HeapRevolver in Linux and Windows and report the evaluation results. The results show that the performance overhead of HeapRevolver is relatively smaller than that of DieHarder [14], which is one of the representative methods to prevent UAF attacks by library replacement.

2 Problem and HeapRevolver design

2.1 Problem of existing methods

The problems of the related studies [15]-[17] are as follows:
(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 complicates UAF attacks. Owing to the period being fixed, attackers can guess the reuse timing. Thus, the reuse time estimation must be made difficult.
(Problem 2) Need to alter the program code: Some methods alter the program of IE and call the recently added functions, thus preventing a UAF attack. Therefore, altering a program is necessary.
(Problem 3) Target application and 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 to resolve these three problems.

2.2 Design of HeapRevolver

In this paper, we focus on the objective that UAF attacks can be prevented by preventing 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 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 reuse in HeapRevolver.

To prevent UAF attacks by reusing the memory objects, HeapRevolver prevents a UAF attack by altering 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 [16] and Memory Protector [17]. 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.

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. Furthermore, HeapRevolver releases at most only half of the freed memory area, implying that the randomly selected memory is delayed. This results in a certain memory area that cannot be reused for a long period. Furthermore, by adding condition 2, a UAF attack fails if an offset of a dangling pointer to the memory area is not appropriately calculated.

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.

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 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.

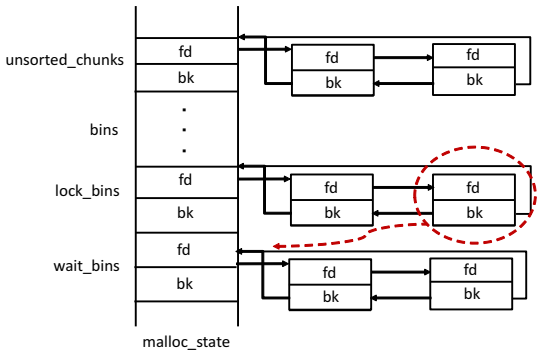

Fig. 2. Memory structure of malloc in HeapRevolver

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 threshold $\geq$ 1 MB. Furthermore, HeapRevolver randomizes the threshold of the total size when the total size of freed memory is larger than the threshold value.

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

The 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. 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. 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

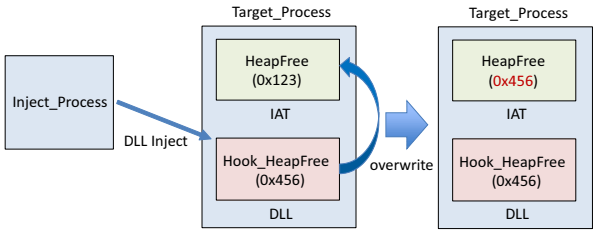

Fig. 3. Flow of hooking *HeapFree()* function on Windows

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.

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 a memory area is already merged with an adjacent memory area. This 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 can complicate UAF attacks using a large number of thresholds and randomizing them.

4 Evaluation

4.1 Security Analysis

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, in this case, reusing the freed memory area is difficult 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 total size of the freed

memory areas. Because the threshold of reuse is randomly set when the freed memory area is released and large-scale programs such as browsers process many memory allocations and releases, 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. In many attacks, the requested size of memory allocation 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. Because such failure in attacks reveals the attempts of attacks, we believe that attackers will avoid performing low-possibility attacks.

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 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, large area of heap spraying is often allocated in the last part of the heap area, and 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, reuse of 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.

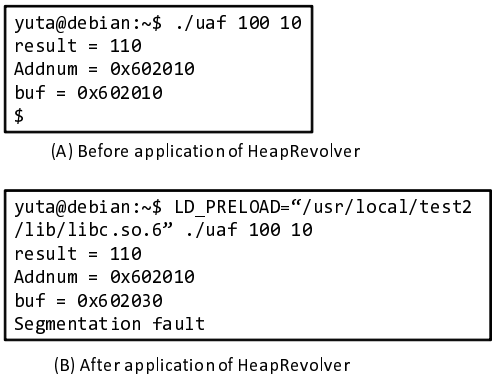

Fig. 4. Experimental results of UAF attack prevention in Linux

To show the feasibility and overhead of the HeapRevolver, we evaluated its performance on Linux and Windows. The following experiments were performed. The UAF-attack prevention experiments in Linux and Windows show 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 performance and memory overhead of HeapRevolver.

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 is created and deleted. Subsequently, when a memory area with the same size as that of the *Addnum* object is created, the memory area of the deleted *Addnum* object is reused. The address where a pointer of the shell code is stored is overwritten on the vtable address of the *Addnum* object. The shell code is executed by a call to the overwritten vtable. The program was executed when 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, Figure 4-(B) shows that an *Addnum* object and *buf* were allocated in different memory areas. Here, the UAF attack failed due to segmentation fault because the memory area accessed by referring to the dangling pointer did not have access rights. Therefore, 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%)

4.4 Evaluation of performance overhead in Linux

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

First, the malloc-test benchmark was used to evaluate the processing time. The malloc-test benchmark contains 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 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 shows that the overhead of HeapRevolver was less than 20% in the malloc-test when the memory sizes were 100 and 512 bytes. The overhead of HeapRevolver 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 show that the large threshold of the HeapRevolver involved large overhead for every requested memory size.

Next, the performance overhead of the HeapRevolver was measured using UnixBench, SysBench and Himeno benchmarks. Table 2 lists the evaluation results, which show that the overhead of HeapRevolver was less than 0.25% in every benchmark evaluation. The performance overhead of the 1-MB HeapRevolver is greater than that of the 100-KB HeapRevolver. We suppose that the performance overhead increases according to the size of the threshold and that the performance overhead is 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. Figures 5 and 6 show the comparison results of HeapRevolver with glibc in Firefox and Chrome respectively, considering their performance overhead.

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%)

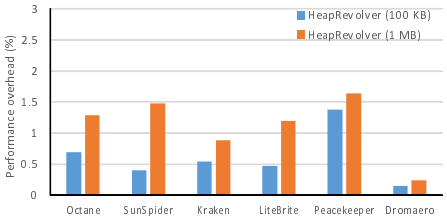

Fig. 5. Performance overhead of browser benchmarks on Firefox

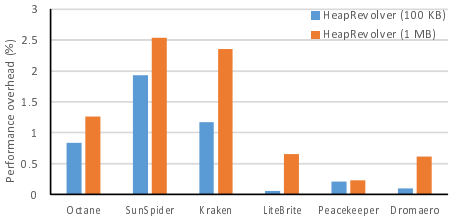

Fig. 6. Performance overhead of browser benchmarks on Chrome

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 duration of reuse was longer, was larger than that in the 100 KB HeapRevolver because the change in the amount of data segment size (heap area), such as *sbrk* system call, increased when allocating a new memory area. Furthermore, Figure 6 shows 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, and ApacheBench was used as a benchmark in measuring the response time of the web server in this evaluation. 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. It shows that the overhead of HeapRevolver in every result was small. However, the overhead of HeapRevolver increased when the requested file size was 0.1 KB. This process included network and CPU processes. Thus, we assume that the overhead of the memory allocation and release were hidden by these processes.

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. In this experiment, five threads were run, and the allocation and freeing processes were performed when the memory size was 512 bytes. Each thread repeated this

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

Method	**Request file size (KB)**			
	0.1	1	10	100
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 %)

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

process 10 million times. We measured the memory usage when the processing of the five threads was finished.

Table 4 lists that the memory usages of glibc and 100-KB and 1-MB 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 show that the maximum overhead of the memory usage for each process is less than the threshold.

We used Firefox 31.0 and Selenium IDE to evaluate the memory consumption when browsing 10 websites continuously. 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 HeapRevolver were almost the same. The memory usage was between 280 and 320 MB because the memory usage overhead of HeapRevolver was small and the variation in memory usage was relatively large.

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

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

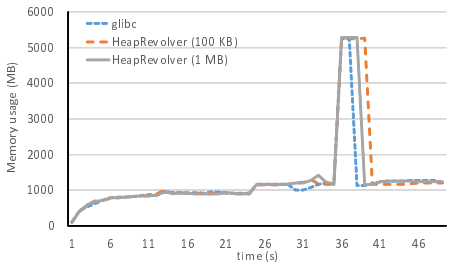

Fig. 7. Memory usage of Octane on Firefox

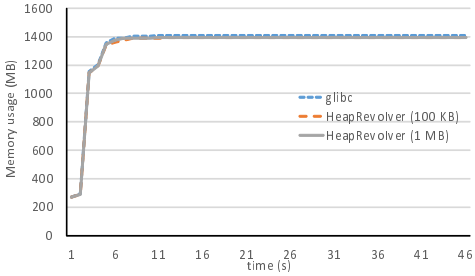

Fig. 8. Memory usage of Octane on Chrome

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 %)

4.6 Prevention experiments against UAF attack in Windows

We experimented on whether UAF attacks using real attack codes distributed in Metasploit could be prevented. The attack codes used in the environments exploited CVE-2011-1260 and CVE-2012-4969 of IE 7 on Windows XP and CVE-2014-0322 of IE10 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 on Windows as described earlier. Then, the attack codes were executed in each environment. Thus, HeapRevolver successfully prevented all the UAF attacks that reused memory objects.

4.7 Evaluation of 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 threshold of HeapRevolver was 3,000. The measured overhead of HeapRevolver in the three browser benchmarks was less than 2.5%. These browser benchmarks are CPU-intensive and require large memory. Thus, we suppose that the influence on the performance of the browser benchmarks can explicitly be observed. Nevertheless, the

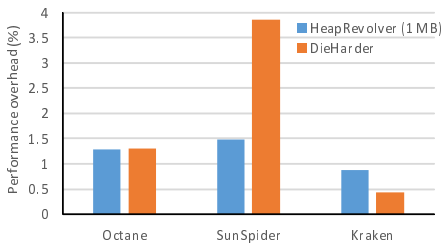

Fig. 9. Comparison of HeapRevolver and DieHarder for browser benchmarks in Firefox

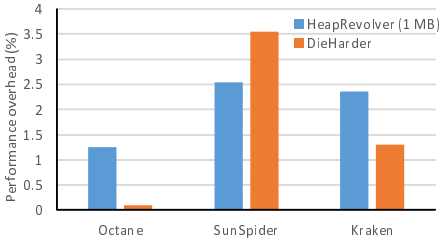

Fig. 10. Comparison of HeapRevolver and DieHarder for browser benchmarks in Chrome

Table 7. 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%)

results show that the overhead of HeapRevolver in Windows is small, and the overhead is acceptable.

4.8 Comparison with existing method

We compared HeapRevolver with DieHarder [14], which can be classified to be the same as HeapRevolver. The threshold of HeapRevolver in this evaluation was 1 MB.

Figures 9 and 10 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 overhead of DieHarder in SunSpider was relatively large (approximately 4%). We will analyze the resultant factor of DieHarder in future; however, we believe some inefficient processing in the reuse of objects in DieHarder occurred.

Table 7 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. In addition, we evaluated the performance overhead results of original glibc using UnixBench, SysBench, and Himeno benchmarks (Table 8). The results show that the performance overhead of HeapRevolver was smaller than that of DieHarder in all benchmarks.

Finally, we evaluated the change in the amount of memory consumption under three browser benchmarks in Firefox. Figure 11 shows that the memory

Table 8. 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%)

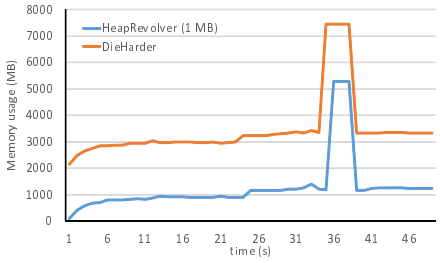

Fig. 11. Overheads of Firefox browser memory usage (Octane)

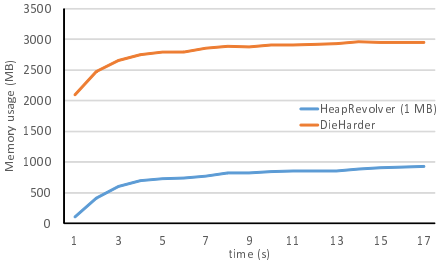

Fig. 12. Overheads of Firefox browser memory usage (SunSpider)

consumption of DieHarder in Octane was more than twice that of HeapRevolver. Figure 12 shows that the memory consumption of DieHarder in SunSpider was approximately three times more than that of HeapRevolver. However, the overhead of DieHarder was very heavy to use in real world. Comparatively, the results show that the memory usage of HeapRevolver was efficient because HeapRevolver delayed the reuse of freed memory within the threshold size.

Next, we 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. Therefore, 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 and that of DieHarder was 87,906,816 KB. These results show that the memory consumption of DieHarder in Octane was approximately twice that of HeapRevolver. In addition, the memory usage trend in Chrome is similar to that in Firefox.

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

5 Related work

Dangling pointer-detection approaches [3]-[7] include dynamic binary translation, shadow memory, and taint analysis. These approaches detect dangling

pointers before program execution. However, if the dangling pointers are abused, which cannot be detected before a practical use, UAF attacks cannot be prevented in runtime. In [8]-[10], UAF attacks were prevented by replacing a malloc library with a new library in which the allocation unit is a page. However, because the allocation unit of the created memory area consists of pages, the memory usage is inefficient. In [11]-[13], a UAF attack was prevented using a method that prevents alteration of vtable. However, these methods cannot handle a UAF attack that does not alter vtable.

6 Conclusions

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

The evaluation results in Linux show that the HeapRevolver overhead is sufficiently small. However, the process of repeating memory allocation and releasing memory slightly influences the performance. Further, the evaluation results show that the increase in the memory consumption is slight compared with that in the original glibc, and the overhead is acceptable. The experimental results in Windows using UAF exploit codes show that UAF attacks can be prevented using HeapRevolver. In addition, the performance evaluation results by using browser benchmarks show that the HeapRevolver overhead is less than 2.5%. Finally, we compared HeapRevolver with DieHarder through evaluations. The results of the browser benchmarks show that the HeapRevolver overhead is smaller than that of DieHarder in most cases and the amount of memory consumption of HeapRevolver is 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 is sufficiently small to be deployed in real systems. We believe that HeapRevolver can prevent UAF attacks by exploiting zero-day vulnerability.

Acknowledgement This research was partially supported by Grant-in-Aid for Scientific Research 16H02829.

References

1. Common Vulnerabilities and Exposures, `https://cve.mitre.org/index.html`
2. Microsoft Security Intelligence Report Volume 16, `http://www.microsoft.com/en-us/download/details.aspx?id=42646`

3. Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D.: Addresssanitizer: A fast address sanity checker, in the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12), pp.309–318, 2012.
4. Caballero, J. et al.: Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities, in the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), pp.133–143, 2012.
5. Nethercote, N. and Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation, in the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), pp.89–100, 2007.
6. Bruening, D. and Zhao, Q.: Practical Memory Checking with Dr. Memory, in 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 213–223, 2011.
7. Lee, B. et al.: Preventing Use-after-free with Dangling Pointers Nullification, in the 2015 Network and Distributed System Security Symposium (NDSS), 2015.
8. GFlags and PageHeap, https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561\%28v=vs.85\%29.aspx
9. Electric Fence, http://elinux.org/Electric_Fence
10. D.U.M.A. - Detect Unintended Memory Access, http://duma.sourceforge.net/
11. Younan, Y.: FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers, in the 2015 Network and Distributed System Security Symposium (NDSS), 2015.
12. Zhang, C. et al.: VTint: Protecting Virtual Function Tables' Integrity, in the 22th Annual Network and Distributed System Security Symposium (NDSS), 2015.
13. Gawlik, R., Holz, T.: Towards Automated Integrity Protection of. C++ Virtual Function Tables in Binary Programs, in the 30th Annual Computer Security Applications Conference (ACSAC '14), pp.396–405, 2014.
14. Novark, G., Berger, E.D.: DieHarder: Securing the heap, in the 17th ACM Conference on Computer and Communications Security (CCS '10), pp.573–584, 2010.
15. Tang, J.: 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/
16. Tang, J.: 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/
17. 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/
18. 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
19. Abdul-Aziz Hariri et al., Abusing Silent Mitigations - Understanding Weaknesses Within Internet Explorers Isolated Heap and MemoryProtection, https://www.blackhat.com/us-15/briefings.html