RLCA : Mitigating Large-Chunk-Based Heap Attacks by Randomization of Large Chunk Allocation

ABSTRACT

Heap security relies heavily on the randomness of chunk allocations in memory allocators to mitigate heap fengshui and heap spray attack, which are the most widely used techniques in modern exploits. However, randomness in large chunk allocation has been overlooked. Memory allocators directly call *mmap* (sometimes *brk*) syscall to allocate large chunks, while the Linux kernel does not provide a fine-grained randomization for *mmap/brk* syscall. Only the base address is randomized, but the offset between every two syscalls is predictable. The less randomized large chunk will be vulnerable to heap fengshui and heap spray attacks.

In this paper, we assess the security of three most representative general-purpose memory allocators, Glibc ptmalloc, OpenBSD PHK malloc, and DieHarder, in scenario of large-chunk-based attacks, with successful heap fengshui and heap spraying attack under Nginx. We then present RLCA, a transparent, portable, and memory allocator agnostic, runtime large chunk randomizer to fortify the existing memory allocators against large-chunk-based attacks. Large chunk fengshui and spraying attacks can be successfully mitigated by RLCA with a fine-grained randomization in *mmap/brk* syscall. RLCA imposes an acceptable overhead in both performance and memory usage.

1. INTRODUCTION

The war in heap security has never stopped. For the last few decades various heap attacks [2] [25] [38] [40] have been proposed. Works on heap attack mitigation [15] [19] [23] [34] [42] and memory error detection [17] [31] [36] [37] are also brought up as countermeasures to heap-based attacks. Among all these mitigation and detection strategies, ASLR [35] [39] is one of the most widely adopted approaches in securing the memory. The appearance of ASLR has raised the threshold for the attackers to develop a reliable exploit. By randomizing the base address of stack and program segments at every load time, the attackers can never exploit the application with an once-and-for-all fixed address.

For all the tries the attackers have made to defeat ASLR, the emerging of heap spray [1] and heap fengshui [38] has become the main stream, and even nowadays, they are still constantly haunting the security researchers. The intrinsic of heap spray is to exhaust as much memory space as possible in purpose of an ASLR entropy reduction, so that the attacker can find a relatively reliable address to land and perform the subsequent attacks. As to the heap fengshui technique, the attacker can predict the allocated chunk address by studying the internal mechanism of the memory allocator, so the attacker can combine any vulnerability, like heap overflow or use-after-free, to corrupt and exploit a controllable chunk.

A finer-grained chunk randomization has proved to be very effective against both heap fengshui and heap spray. Security memory allocators [30] [32] are proposed to facilitate a more randomized chunk allocation in a design of BiBOP [22] style, with "Big Bag of Pages" acting as a memory pool. However, for all the security memory allocators, we have found that the randomization is partial. Chunks are only randomized within a manageable size, which is usually 1 page, since the randomize-able chunks are all organized into single pages. For chunk of size over 1 page, a *mmap* syscall will be invoked directly, and the allocation of such chunk is handed to the system to ensure its randomness. The problem is that the *mmap* syscall of the underlying system is not fully randomized. In Linux kernel, *mmap* is only randomized in the base address, but the relative offset of each *mmap* syscall is not randomized. Every *mmap* syscall maps the memory linearly down from the base address. It will lead to a potential vulnerability that such less randomized memory space can be leveraged by heap fengshui and heap spray.

To define a large chunk in a general environment, we consider large chunks as those allocated directly with *mmap* syscall in the memory allocators. The defined size of large chunk varies with different memory allocators. For security memory allocators, the size is mostly set as 1 page (4KB), which is the case of OpenBSD PHK malloc [30] and DieHarder [32]. Other general-purpose memory allocators have more diversed definitions. For instance, glibc ptmalloc [21] draws the line at 128KB, while dlmalloc [27] is 256KB. Application-specific memory allocators are those implemented internally in large applications. They either reuse system malloc to handle large chunks or directly use *mmap* syscall for the large chunk allocations. Nginx internal memory allocator [33] reuses system malloc for chunk size over 1 page, which makes the large chunk allocation exactly

the same as the general purpose memory allocators. Php zend allocator [9], on the other hand, handles chunk size over about 2MB through direct *mmap* syscall.

However, there is no effective mitigations against large chunk spraying and large chunk fengshui attacks. Guard page [5] is one technique that can be used to mitigate heap spraying attack, though it is originally widely adopted as a protection against heap overflow attack. For every large chunk allocated, a page with no privilege will be appended after the chunk, and any access to the page will trigger a SEGV fault. It can reduce the success ratio in a heap spray attack by a limited amount, but still far from a sound mitigation. We will show in Section 4.2.2 that it is trivial to bypass guard pages in a heap spraying attack. Graffiti [19] provides an empirical solution to detect heap spray attack, but suffers from false positives and is limited to Intel CPUs with certain virtualization feature. PaX RANDMMAP [8] and ASLP [24] randomizes every *mmap* allocation so as to mitigate both heap fengshui and heap spray, but, on the other hand, restricted to the portability, since they both require to patch the kernel to enable such feature, and in functionality, both of them disable *MAP_FIXED* in *mmap* syscall which will sometimes cause troubles (Section 5). Still other fine-grained ASLR solutions [13] [14] [41] randomize the placement of every chunk by padding a random size either at heap base address or on every allocation. This kind of mitigation is far from enough to successfully mitigate heap fengshui and heap spray, either. Random padding can be easily bypassed with a large heap overflow in heap fengshui attack, and the padding can also be negligible in the heap spraying attack as long as the chunk sprayed is large enough.

To overcome the problems of the previous works, we present RLCA to mitigate both large-chunk-based heap fengshui and heap spray attacks by randomizing large chunk allocations at runtime. RLCA provides a comprehensive randomization to the large chunk with an optimal overhead. We design RLCA to protect large chunks only. The small chunk randomization is left to the security memory allocator, like DieHarder, and PHK malloc, since they have already done extensive and excellent work in this field.

At the core of RLCA is the memory profiling structures representing the whole memory layout. We design the data structures with full redundancy so that it can provide an optimized performance in either allocation/deallocation operations or de-fragmentation operations. The structures are also designed to provide less fragmentation by reusing small memory holes for single page allocations.

We design RLCA as a transparent layer between the system and the memory allocator to ensure large chunk randomization regardless of the types of the memory allocator. For the syscall intercepted, we follow the same handling principles as the Linux kernel in unexpected parameters. We provide a fine-grained randomization spreading through the whole virtual memory address space by intercepting all the *mmap* syscalls with an efficient binary instrumentation framework, and choosing an address randomly through the memory profiling structures.

There are cases that large chunk will also be allocated through *brk* syscall, which is normally used for small chunk allocations in some general purpose memory allocators. We handle this problem as well by intercepting the *brk* syscall at runtime and mimicking *brk* with a preallocated memory space which can be moved swiftly as the memory space can no longer hold the coming requested size. This strategy can also make the *brk* region less predictable against large chunk spray attacks in some extreme conditions when large chunks are allocated in the *brk* region.

In the evaluation, we designed large chunk fengshui and large chunk spraying experiment based on CVE-2014-0133 and CVE-2013-2028, under glibc ptmalloc and OpenBSD PHK malloc. Our result shows that RLCA can successfully mitigate both large-chunk-based attacks with a fine-grained randomization strategy. RLCA also introduces little performance penalty. The runtime performance overhead of RLCA is less than 10% on Nginx. We test the single execution performance of RLCA against a list of most widely used applications, and the result turns out an average overhead of less than 20% in performance and 2% in memory usage.

The contributions of our paper are listed as follows:

1. We present RLCA, a portable, memory allocator agnostic runtime address randomizer for large chunk allocation.

2. We provide a thorough security analysis for OpenBSD PHK malloc and DieHarder, and a technique to leverage security memory allocators through large-chunk-based attacks.

3. We develop the first working attack on CVE-2014-0133 Nginx SPDY heap overflow which can be further used for security researches.

The rest of the paper is organized as follows. In Section 2, we present the threat model and the detailed analysis of three representative general-purpose memory allocators to illustrate the attack surface introduced by insecure large chunk allocations, as well as some backgrounds in heap protection. Section 3 provides a detailed design and implementation of RLCA. Section 4 shows the evaluation of RLCA in both security and performance. Section 5 is the discussion, and Section 6 is the related work. Section 7 is the conclusion.

2. LARGE-CHUNK-BASED ATTACKS & MITIGATIONS

In this section, we illustrate how to attack the large chunk in real-world memory allocators, and then introduce the mitigation background. Our attacks are based on the following assumptions:

- The target program is deployed on conventional Linux distributions (e.g. Debian, Ubuntu) using native Linux kernel. The type of glibc or the underlying memory allocator is not restricted, as long as the attacker is able to allocate large memory chunks, and is able to trigger *mmap/brk* syscall by this allocation.

- The attacker can attack the application through heap spray and heap fengshui. And, she has at least one vulnerability that can either corrupt the memory or hijack the control flow. Because both heap spray and heap fengshui are just vehicles for the attacks, which are used to facilitate the attacker to develop a reliable exploit, and they don't have the ability to corrupt or hijack the program, at least one vulnerability is required to trigger the attack.

- There is no information leak in the target program or system, because if there is any, heap spray or heap fengshui would be meaningless. Information leak breaks ASLR by intrinsic. The attacker can only land to a guessed address in a reliable memory padding through heap spray, or corrupt certain critical memory structures by learning the heap memory layout through the studying of the memory allocator as is done in heap fengshui. We define such critical memory structures as heap metadata, which is a common attack surface of memory allocators.

2.1 Large-Chunk-Based Attacks

To illustrate how large chunk allocations are vulnerable and how they can be exploited, we analyze the internal mechanism and attack surfaces of three representative memory allocators: glibc ptmalloc [21], OpenBSD PHK malloc [30], and DieHarder [32]. We will illustrate how we can possibly exploit the large chunk allocation in real world in this section. Glibc ptmalloc is a memory allocator widely used in the Linux distributions, including Debian, Ubuntu, etc. OpenBSD PHK malloc and DieHarder are two representative security memory allocators that can be ported to any system to help fortify the application. OpenBSD PHK malloc is originally a memory allocator implemented in the OpenBSD operating system. However, many have ported this implementation to Linux distributions because of its security. DieHarder is an even safer memory allocator that introduces a finer grained randomization and a complete heap metadata discretion.

2.1.1 Glibc ptmalloc

Glibc ptmalloc is a free-list based memory allocator. Every chunk stores a header to indicate the address of the previous or next chunk. Large chunks are defined as chunks larger than *mmap_threshold* (128KB by default). For large chunk allocation either *mmap* or *brk* will be called. If the total *mmap* allocation is less than *n_mmaps_max* (65536 by default), *mmap* will be invoked for large chunk allocation. Otherwise, *brk* will take the place instead. Small chunk allocations in ptmalloc are performed through a large continuous memory space called "arena". Instead of focusing on the randomization of chunk placement, ptmalloc focus on chunk reuse to provide a better performance.

Heap fengshui and heap spray attacks are trivial. Since either *mmap* or *brk* allocates memory in a continuous way, requesting large chunks constantly in the spraying attack will always result in a large memory space full of attacker's payload. Also, since heap metadata is placed as a header along with the chunk data, and any overflow of the chunk will lead to a heap metadata corruption, heap fengshui is only needed to place a controllable chunk right after the overflowed one, so that the heap metadata of the controlled chunk can be manipulated directly. A lot of heap overflow exploitation techniques can also be found both in publications and wild. [20] [10] [6]

2.1.2 OpenBSD PHK malloc

OpenBSD PHK malloc differentiates chunk allocations with size less than or equal to 1/2 page, and greater than half page. *mmap* is directly called for large chunk allocation. For small chunks, chunk size ranges from 16 bytes to 2048 bytes with an alignment of 2^n where 4 <= n <= 12. Every chunk of the size will be allocated in a private pool of 4 pages. On allocation, 1 of the 4 pages will be selected randomly and a random offset will be generated to get the new chunk.

Chunk metadata (See Code 1) is created and managed separately from the chunk data. *region_info* is one structure used to keep track of the chunk addresses. It is managed in an array with a random placement. Every time a large chunk is created, a new *region_info* entry will be randomly placed and updated in the array. For small chunk, *chunk_info* structure is created for every page in the pool to record the chunk usage. A *region_info* structure will be added accordingly as well for every *chunk_info* structure.

Listing 1: OpenBSD PHK malloc Internal Struct Code Snipets

```
struct region_info {
    /*page; low bits used to mark chunks*/
    void *p;
    /*size for pages, or chunk_info pointer*/
    uintptr_t size;
#ifdef MALLOC_STATS
    void *f;  /* where allocated from */
#endif
};

struct chunk_info {
    LIST_ENTRY(chunk_info) entries;
    void *page;  /*pointer to the page*/
    u_int32_t canary;
    u_short size;  /*size of this page's chunks
        */
    u_short shift; /*how far to shift for this
        size*/
    u_short free;  /*how many free chunks*/
    u_short total; /*how many chunks*/
    /* which chunks are free */
    u_short bits[1];
};
```

For the heap fengshui attack, the attacker can successfully control heap metadata by overflowing the *region_info* array. The *region_info* array is the only heap metadata without a forced guard page. It is directly mapped with *mmap* syscall, and it will grow in a multiplier of 2 to ensure randomness, if the free slots is less than 1/4th of the whole size. The attacker can allocate a large chunk right after a growth in the array, and overflow the large chunk to take full control of all the *region_info* structures. To circumvent the random placement of *region_info* in the array, the attacker can overflow as many entries as possible, so there will be a higher probability to trigger the crafted *region_info* when manipulating heap chunks.

To exploit heap spray, the attacker can simply trigger the large chunk allocation for many times. The allocated chunks will be mapped directly through *mmap* syscall, and the attacker's payload will be sprayed into a large continuous memory space, leading to the ASLR entropy reduction.

2.1.3 DieHarder

DieHarder is a representative memory allocator in the DieHard series. It performs similar strategy as PHK malloc in chunk allocations. For large chunk over one page, *mmap* will be invoked, and for small chunk, a randomized allocation is performed. Small chunks of size from 8 bytes (for x86) or 16 bytes (for x64) to 1 page (4096 bytes) are each aligned by a power-of-2. (Figure 1) Every chunk of the

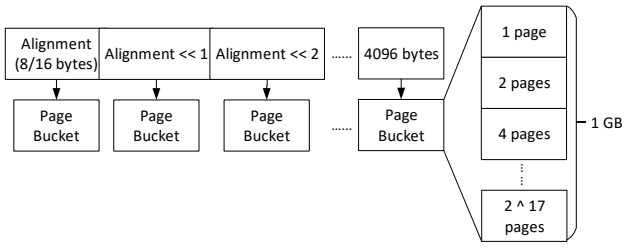


Figure 1: DieHarder Internal Memory Structure

size is organized into pages as a private memory pool, with a bitmap structure, which is allocated dynamically by the underlying system malloc, to store the chunk usage information. Chunk pages are managed by an array of 18 entries for an incremental page request. Each entry takes charge of 2^n pages ($0 <= n < 18$), making a maximum of 1GB for every chunk of the size. Whenever the number of free chunks is less than a certain threshold, the next entry will be "activated" to request extra pages, so the randomness requirement can be satisfied.

The randomization procedure splits up into two stages. At the first stage, DieHarder maps a large memory pool to handle page requests randomly. Stage two randomizes the chunk selection. DieHarder randomly chooses a page from the activated ones, and keeps randomly probing until an unused slot is found.

Since DieHarder declares the heap metadata with "static" keyword, it is impossible to overflow the metadata because they are all stored in data segment, and the size of heap metadata is defined and fixed at compile time, so the attacker cannot manipulate the location of heap metadata through heap fengshui. The only metadata that is allocated dynamically at runtime is the bitmap structure. However, there stands a slim chance to attack the bitmap with large chunks. So, we consider DieHarder invulnerable to attacks overflowing heap metadata.

However, DieHarder is still vulnerable in large chunk spraying attack. DieHarder also directly uses the system *mmap* to handle large chunk allocation. The exploiting strategy is identical to the OpenBSD PHK malloc.

2.2 Mitigations

Aside from the native security memory allocator implementations, various heap protection methods have been proposed to mitigate heap based attacks.

Those protections can be categorized into three categories: DBI assisted, compiler/kernel/hardware assisted, and native hook.

2.2.1 DBI Assisted

DrMemory [17], Memcheck [37] are built based on DBI (Dynamic Binary Instrumentation) tools in purpose of memory error detection. They instrument and intercept the malloc/free calls, and trace the allocation and deallocation of every chunk. Both are designed to be able to detect various memory errors, including heap overflow, malicious heap free, UAF, etc. However, they are too comprehensive, and too slow to be implemented in real-time services. The average performance overhead is 15~30 times. They cannot defeat heap fengshui and heap spray either, if triggered through logical vulnerabilities, like type confusion vulnerability.

Vivek Iyer et. al. [23] proposed a solution using Detour to dynamically hook malloc/free functions and place random paddings around the allocated chunk. However, it still fails to provide a throughout protection against large-chunk-based attacks. For either large chunk fengshui or spraying attack, the random padding is too small in size and is negligible comparing to the large chunk. It also fails to protect x64 system, because Detour only supports x86 binaries.

2.2.2 Compiler/Kernel/Hardware Assisted

Graffiti [19] relies on EPT page-table, which is a virtualization feature of some Intel processors, to detect heap spraying attack. HeapSentry [31] requires to insert a kernel module to facilitate heap overflow protection. CRED [34], AddressSanitizer [36], HeapTherapy [42] require source code to perform compile-time instrumentation. Those approaches can provide a very low performance overhead with a relatively comprehensive mitigation. However, those approaches are not portable, because they either require source code or specific hardware features or kernel support to fully mitigate heap based attacks.

2.2.3 Native Hook

Runtime hooking can protect the heap by fortifying the malloc/free functions with LD_PRELOAD or glibc ptmalloc native hooks. The former one hooks by hijacking the symbol resolution in the glibc loader (ld.so), while the latter one requires a native implementation with source code. For most of the work, LD_PRELOAD is perferred because of its portability. It will work as long as suitable dynamic library is provided. Heap protection of this kind either provides a wrapper for the malloc/free functions [26], or re-implements a much safer memory allocator [30] [32] [12] [11].

Wrapper approach is restricted to tracing and appending heap red-zones only. The memory allocation still reuses the system memory allocator.

Re-implementation approach is not sound either. Since LD_PRELOAD only hijacks the symbol resolution of library function calls, invoking syscalls directly in the application will escape the protection. This concern is reasonable because nearly all the large applications will implement their own internal memory allocator for better performance. The injected library will fail to catch the large chunk allocation call if the internal memory allocator involves a direct *mmap/brk* syscall. This will lead to unsafe memory allocations as well.

3. DESIGN & IMPLEMENTATION

We designed RLCA to provide a light weight protection for large chunk allocations in both general purpose memory allocators and application specific ones. We directly intercept the syscall instructions and provide a fine grained randomization by an efficient binary instrumentation. On occurrence of a *mmap* syscall instruction, we will try to find a randomly selected address, and modify the syscall parameters accordingly. We do not override the *MAP_FIXED* flag in order to make RLCA transparent to the application. For *brk* syscall, we emulate a fake *brk* by preallocating a memory space at the first time. We would expand or move *brk* region adaptively when the size requested exceeds the preallocated memory. In this section, we will show the design overview, and some design concerns, and then we will discuss

the implementation detail of RLCA.

3.1 Design Overview

RLCA provides randomization for large chunk allocations at runtime. The basic idea is to intercept all the *mmap* and *brk* syscalls and redirect them to a safer implementation. In the design of RLCA, we aim to satisfy the following requirements:

- *Portability* : RLCA should be portable and easy to deploy.
- *Comprehensiveness* : RLCA should be able to protect any memory allocator from large-chunk-based attacks. Not only *mmap* but also the less used *brk* should be intercepted to provide a comprehensive protection throughout the whole life-cycle of the program.
- *Transparency* : RLCA should be totally transparent to both the upper layer application and the underlying system. The functionalities and features of the underlying system should be preserved, and the application can be safely ported to RLCA without any modification.
- *Fine-Grained Randomization* : RLCA should provide a fine grained randomization at runtime that the entire virtual memory address space should be utilized.
- *Acceptable Overhead* : Overhead introduced by RLCA should be acceptable.

In principle of that, we build RLCA as a client library on top of DynamoRIO [18], which is a transparent instrumentation framework with little runtime overhead. It is portable in that RLCA is a pure user land approach that directly manipulates the application binary and does not require any kernel or hardware support. We inject RLCA library at program load-time and provide a throughout tracking of all the *mmap* and *brk* syscalls to make it comprehensive.

To provide a fine grained randomization for *mmap* and *brk*, we should first have a general idea of the process memory layout. We obtain this from /proc/*pid*/maps file. We store this information in the memory profiling structures in avoidance to repeatedly read and parse the maps file every time we perform an allocation.

One concern is the memory usage management. Since RLCA works as a library injected into the target program's memory space, it shares the same address space with the application. We should be noted that any dynamic memory allocation from RLCA may interfere with the whole memory layout of the application. Although DynamoRIO implements internally a preallocated memory pool for dynamic memory use, allocating dynamic objects still needs precaution, since we cannot intercept the memory allocation action from RLCA itself. In the design and implementation of RLCA, we try to avoid any dynamic data structure to reduce the dynamic memory usage.

Instrumentation also requires to allocate dynamic memory. We consider this fine to use the DynamoRIO internal memory allocator, because the life-time of instrumentation instructions exists only within one basic block, and, instrumentation takes only a small memory, which will be covered by the DynamoRIO preallocated memory pool safely from corrupting the memory space layout.

For multi-threaded applications, we add lock in RLCA to avoid race condition.

3.2 Implementation

We implement RLCA to be a light-weight and transparent randomizer that mimics the *mmap* family and *brk* syscalls to provide a transparent layer between application and operating system. At the core of RLCA is the memory profiling structures. RLCA will first tries to identify instructions of interested syscall: *mmap*, *munmap*, *mremap*, and *brk*. Initialization of the memory profiling structures (by parsing the /proc/*pid*/maps file) will be triggered once these instructions are met for the first time. Memory profiling structures will facilitate the allocation and deallocation of memory space. Any mapping or unmapping operations will update the memory profiling structures as well. De-fragmentation procedure will be performed at times to keep memory profiling structures consistent.

3.2.1 Memory Profiling Structures

To start with, we implement *AddrRange* structure to store all the unmapped memory ranges so as to represent the whole memory layout. It is initialized as an array, so the memory usage by such management data can be controllable. Each element in the array holds the start and end address of one unmapped virtual memory range.

Randomizing large chunks of relatively small size, e.g. 1 page in security memory allocator, will cause severe fragmentation if those small-sized large chunks are also placed randomly throughout the whole memory space. We try to reduce such fragmentation by reusing the small memory holes and pages at the margin in the memory layout. *PageSlot* is thus designed as a page pool to manage all the single page dispositions. *PageSlot* is also initialized as an array. Every time we find a memory range of size 1 page, an entry will be added into the *PageSlot*. We keep the *PageSlot* unsorted, so the single page operation will be executed in O(1) time. For the single page allocation, we will randomly select a page from *PageSlot*. We guarantee the minimum randomness of 16 by filling the *PageSlot* from *AddrRange*. The filling starts from the least sized entry in *AddrRange*, splitting each into single pages, and fills the *PageSlot* until it reaches 16. The remaining of the entry will stay unmoved in *AddrRange* as long as its size is larger than 1 page. We tend to keep *AddrRange* free from single page allocations, so all the memory range of 1 page will be moved to *PageSlot*.

AddrRange is sorted by the start(/end) address so the defragmentation procedure can be extremely efficient (in complexity of O(N)). However, a design of such is still under optimized for memory allocation. It will have to go through all the entries in *AddrRange*, pick out ones with suitable size, and randomly select one from these candidates. The minimum complexity will be O(N). We managed to reduce the complexity to O(log(N)) by introducing another array, *SizeList*. Each element in *SizeList* matches one in *AddrRange*, only that it is sorted by size. Each entry in *SizeList* has a ref field indicating the index of the corresponding entry in *AddrRange*. Also, each entry in *AddrRange* has a field to reference the index in *SizeList*. Figure 2 shows the overview of memory profiling structures.

3.2.2 De-fragmentation

We will de-fragment the memory profile structures some-

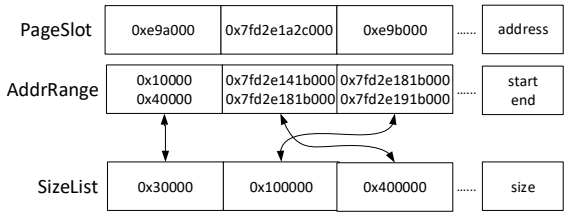


Figure 2: Architecture of Memory Profiling Structures (AddrRange, SizeSlot, PageSlot)

time for a better performance and randomization. We design Garbage Collection procedure to perform this task and rearrange the entries. We provide 4 types of garbage collection strategies for a more refined granularity in performance control. *ReleaseRange* procedure provides interface for *AddrRange* entry merging. This is used to handle unexpected *AddrRange* fragmentation in one continuous memory space. *ReleaseSlot* procedure provides interface to merge *PageSlot* pages back to *AddrRange*. This is used exclusively to release *PageSlot* entries when it is full. *ReleaseSlot* will, however, keep the minimum randomness of 16 pages in the *PageSlot* for the next allocation. Aggressive Garbage Collection cleans up all the *PageSlot* entries regardless of the minimum randomness requirement, and it will clean up *AddrRange* structure accordingly. The final attempt in garbage collection is to re-parse the process maps file, and re-initialize the memory profiling structures.

A Garbage Collection procedure will be invoked when any of the memory profiling structure is full, or when we failed to find a suitable entry for the requested memory range.

3.2.3 Deallocation

When an unmapped memory range is found, or when *munmap* is called, a new entry should be created and inserted into the memory profile structures. We implement *InsertEntry* procedure to perform this task. It will determine if the size is one page or multiple pages. For single page, the procedure will lead to *PageSlot* manipulation, and for multi-page, it will go for *AddrRange* instead. Either way, *InsertEntry* will first check if the corresponding structure is full. A new entry will be added directly if not full. Otherwise, *ReleaseSlot* or *ReleaseRange* will be called to merge the entries. If garbage collection has performed successfully, we will have a safe insertion, with either freed entries, or the inserted memory range has been successfully merged into the structures. However, if garbage collection fails, a force insertion procedure will be invoked. This procedure will keep the structure size unchanged, and insert the new entry by removing the old ones. For single page, the force insertion procedure will perform a random replacement of the old entries. For multi-page force insertion, we will first try to split the *AddrRange* entry of the least size into *PageSlot*, and then fill the place with the new entry. If the removed entry fits fine into the *PageSlot*, nothing will happen because it is a safe insertion. Otherwise, an entry drop is inevitable. We will try to fill the *PageSlot* as much as possible from the beginning of the entry to reduce the entry drop loss, and discard the rest. At the end of the force insertion procedure, a global variable will be set to indicate any drop of the entry.

3.2.4 Allocation

Single page allocation performs directly through *PageSlot*. For multi-page allocation, we will search the *SizeList* for the suitable entries first, and randomly select one for the allocation. Since every entry has a cross reference in both *AddrRange* and *SizeList*, we can update the structures very efficiently. If we fail to find a suitable size, we will perform the aggressive garbage collection to de-fragment, and we will then try again to allocate.

In case we still cannot find a suitable size even after the garbage collection, and we have performed entry drop before, we consider the drop action may be the cause of the failure, and our memory profiling structures may not be large enough to hold the whole memory space layout. We will re-initialize the memory profiling structures, and expand their size with a multiplier of 2 if necessary.

3.2.5 Implementing *mmap/mremap/munmap*

We implement similar *mmap/mremap/munmap* strategy as in the Linux kernel to handle malicious operations.

For *mmap*, a difference comes from the exception we made for single page allocations. We first normalized the input address and the input size by aligning them to page size. Then we check if the requested size is single page or multi-page. For single page mapping, the minimum randomness of PageSlot should first be satisfied. If *MAP_FIXED* flag is set, we will search through both PageSlot and AddrRange for the specific address. We conclude any failure in finding the address as either a malicious map operation or a developer's hack, or just a glitch caused by entry drop. We will tolerate all the conditions and return *DIRECT_PASS* flag to indicate this condition. RLCA will directly pass the original parameters to the underlying syscall. If *MAP_FIXED* flag is not set, we will simply randomly choose one from the *PageSlot* (since the minimum randomness is already ensured at previous step).

For multi-page allocation, and with *MAP_FIXED* flag set, we will again search the exact address in *AddrRange* and try to find the exact entry (or the closest one). Then, we check if the requested range places perfectly inside the entry. If yes, it is fine to just split the entry and return the address. However, if we have a bad fit, a possible cause may be the missing pages placed in *PageSlot*. We will eliminate this possibility with one more aggressive garbage collection to rearrange the memory profile. We will try the allocation for a second time, and if the second try fails again, we will thus have the same scenario as in the single page allocation that the overlapped parts will be overwritten (including the page privilege) by the new mappings. So, we will simply update the memory profile accordingly, and return *DIRECT_PASS*. For allocating multi-page without *MAP_FIXED* flag, we will find the smallest suitable-sized entry from *SizeList*, and randomly choose one from all the entries with size larger than the requested size. We randomly split out the requested size from the selected entry as the return value.

For *munmap*, we will insert the memory back to the profiling structure. Before the insertion, we should check if there are malicious un-mapping operations. If any part of the requested memory overlaps with an unmapped address, the un-mapping procedure should fail.

For *mremap*, we will expand directly or move the old address to a random address or a fixed new address. Different operations are flagged by different combinations of

MREMAP_FIXED flag and *MREMAP_MAYMOVE* flag. If *MREMAP_FIXED* and *MREMAP_MAYMOVE* are both set, we should move the old address to a fixed new address. The Linux kernel handles overlapped new address like in the *MAP_FIXED* case of *mmap*. In this situation, we will insert back the old address and remove the corresponding entries of the new address. If the *MREMAP_MAYMOVE* is solely set, we will unmap the old address, and map a new memory space randomly. If none of the flag is set, which means we are in the direct expand case, we will then try to directly allocate the memory from the end of old address. In all the conditions, the old address should be checked to be fully mapped (which means it does not contain any unmapped pages). Any condition not mentioned above should fail.

3.2.6 *Implementing brk*

Although *brk* is seldom used in large chunk allocations, we still try to protect it out of comprehensiveness. Different from *mmap*, *brk* starts from a more predictable address, and it is designed to grow in a contiguous memory space. Our design has to consider both the *brk* features and the security requirements for randomness. RLCA keeps the continuous memory allocation of *brk*, but randomizes the base address dynamically. We achieve this by preallocating a memory space in initialization. Every time we receive a *brk* syscall, we will directly move the *brk* end pointer along the preallocated memory to mimic a *brk* allocation. If the *brk* address requested exceeds what we have allocated, we will try to expand directly or move the old *brk* memory space to a new address. If either one fails, we will simply follow the native failure procedure of *brk*. To directly expand *brk* memory in place, we just perform the *mmap* style search for the incremental memory. When an appropriate address is found, this space will be mapped, and the *brk* memory region will be updated. However, in case direct expand fails, we will try to find a new place large enough for the new *brk*. We can find a suitable place by the same *mmap* routine. The problem is that we will have to fix all the pointer references to the old memory space to the new location, when we move the *brk* data.

To solve this problem, we implement a small taint tracking engine (Figure 3) to keep records of all the pointer references to the *brk* memory once it has been allocated. We handle registers and memory references differently in taint tracking engine. Registers are placed per thread in the preallocated threadlocal memory space. Each register contains the taint value assigned to it recently. Memory references are stored in Key-Value pairs. Referenced *brk* addresses are the keys, and the referencing memory addresses are the values. Rather than implementing this into a hash map, we turn to a more memory efficient solution with two arrays to store taint values (key) and memory references (value) separately. Each memory reference has a ref key pointing to the entry of taint value, and, each taint value has a counter recording the number of references. Taint addition, taint deletion, and taint population are performed through entry manipulations.

To improve the performance of taint tracking, we only track the memory access operations. We check every write operation if the value is within the *brk* region, or the destination address has already been tainted. We will add, remove or populate taints in different situations. This improvement

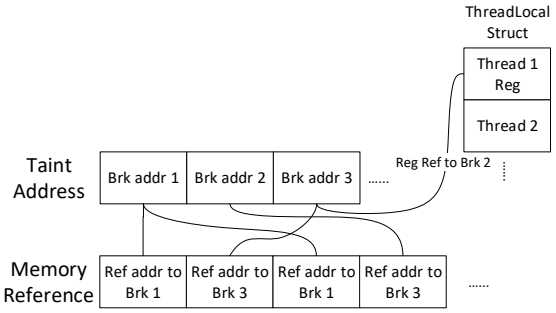


Figure 3: Brk Taint Tracing Structures

(/simplification) proves quite robust in our experiments.

We have also observed that memory allocator, which falls to *brk* to allocate large chunks, will always prepare a failsafe strategy that if *brk* fails to place the large chunk requested, *mmap* syscall will be invoked for the future large chunk allocations. To further optimize the performance, we add an upper bound to the maximum *brk* size so that we can tune the upper bound to control the taint table size, which will greatly improve the overall performance, by preventing large chunks to be allocated in the *brk* region, and the subsequent large chunks will all be allocated through an efficient *mmap* routine without taint tracking. We will show in Section 4.3.2 that tuning the upper bound can greatly improve the performance.

4. EVALUATION

Our experiment is based on the Nginx 1.4.7, under x64 Ubuntu 14.04 LTS with kernel version 3.13.0-89-generic, without the CVE-2014-0133, SPDY heap buffer overflow, and CVE-2013-2028, stack buffer overflow, patch. There are a lot of publications [16] and open exploits [2] for CVE-2013-2028, so we will not discuss too much detail of this vulnerability. Since there is no open exploits or POCs for CVE-2014-0133, we developed our own script to exploit it, and we present a detailed analysis in the Appendix A.

Listing 2: ngx_http_spdy_read_handler Code Snipets

```
1  smcf = ngx_http_get_module_main_conf(sc->
       http_connection->conf_ctx,
2      ngx_http_spdy_module);
3
4  available = smcf->recv_buffer_size - 2 *
       NGX_SPDY_STATE_BUFFER_SIZE;
5
6  do {
7      p = smcf->recv_buffer;
8
9      ngx_memcpy(p, sc->buffer,
           NGX_SPDY_STATE_BUFFER_SIZE);
10     end = p + sc->buffer_used;
11
12     n = c->recv(c, end, available);
```

We perform all our attacks on a single worker Nginx server in avoidance of any glitch in multi-worker environment when trying to manipulate the memory layout precisely.

4.1 Vulnerability Detail

CVE-2013-2028 is a stack-based overflow caused by the incorrect handling of http packet. Stack frame can be directly

corrupted and the attacker can hijack the control flow by overflowing the function return pointer lies in the stack.

CVE-2014-0133 is a heap-based overflow in the Nginx SPDY protocol. As shown in Code 2, before the calling the recv at line 12, a size will be added (line 10). However, *buffer_used* can be set to a large number with a properly crafted SPDY packet, which will eventually cause the overflow in *recv_buffer*.

We find CVE-2014-0133 perfect for our experiment because the overflow takes place at a very large chunk, approximately 256KB. Allocation of this chunk will escape the internal memory allocator of Nginx, and directly call the system malloc. From the analysis of memory allocators in the Section 2.1, this large chunk will either be allocated with *mmap* syscall or *brk* syscall. Either approach will fall into the vulnerability of less randomization.

4.2 Security Evaluation

We have designed three types of attacks to leverage the glibc ptmalloc and OpenBSD PHK malloc: heap fengshui attack, heap spray attack, and brute force enumeration attack. Heap fengshui attack is performed in both PHK malloc and ptmalloc to evaluate the effectiveness of RLCA in protecting security memory allocator and insecure memory allocators. Heap spray attack and brute force enumeration attack are evaluated together, under PHK malloc and ptmalloc as well, to evaluate the effectiveness of RLCA in mitigating entropy reduction attacks under both *mmap* and *brk* memory space. The result is shown in Table 1.

We do not include DieHarder in our experiment because: (1) The heap metadata of DieHarder is statically placed in the data segment, and it cannot be manipulated through heap fengshui attack; (2) DieHarder presents identical feature as OpenBSD PHK malloc in heap spray attack.

4.2.1 Heap Fengshui Attack

In this attack, our aim is to manipulate the memory layout through heap fengshui and overflow the heap metadata by CVE-2014-0133 [3]. The overflowed buffer, *recv_buffer*, is directly *mmap*ed in both of the memory allocators. Attacking ptmalloc chunk metadata is trivial. Since the metadata places along with the chunk data as a header, any overflow can somehow corrupt the heap metadaata. Only a little manipulation is required to make the overflow corrupt a chunk we can control. To exploit this, we start another thread requesting through the http fastcgi routine before we trigger the vulnerability. This will pad a chunk before the overflowed one. After the overflow, we can free this chunk by closing the http connection.

For OpenBSD PHK malloc, additional chunk paddings are required to force the heap metadata to reallocate. Theoretically, we need to allocate $512 * 3/4$ large chunks before the reallocation of heap metadata, but there are adjustments have to be made because of the previous large chunks allocated in Nginx. We place the vulnerable buffer right after the reallocation of heap metadata, and trigger the overflow to take control of the metadata.

In the experiments, We get reliable overflows on both glibc ptmalloc and OpenBSD PHK malloc with RLCA protection off. Both of the heap metadata are corrupted. For glibc ptmalloc, the size field of the next chunk is corrupted, so the attacker can maliciously unmap a memory region of any size when freeing this chunk. For OpenBSD PHK malloc, the *region_info* structs are overwritten by attacker controlled data, so the attacker can intentionally free any chunk or clear the *region_info* of certain chunk. Thus, the attacker can either unmap a memory region, or transform the overflow into an UAF attack, leading to potential code execution or information leak.

When RLCA is turned on, both of the attacks fail to overflow the heap metadata because of the *mmap* randomization. In most cases, SEGV fault is generated because the overflow writes to an unmapped address.

4.2.2 Heap Spray Attack

In this attack, we focus on heap spraying to reduce ASLR entropy. The entropy we can successfully reduce depends on the size of the chunk we can spray and the number of chunks we are allowed to allocate. In Nginx, luckily, both can be controlled through the config file. Chunk size we spray depends on *client_body_buffer_size*, and number to allocate are limited by *worker_connections*. We utilize the CVE-2013-2028 [16] Stack Overflow vulnerability to hijack the control flow with a stack pivoting gadget. We spray our payload with large chunks, and put our crafted stack into the memory. For every 1024 bytes we sprayed, we pad a long chain of ret gadget in the front, and append the ROP chain at the end. This will greatly improve the success ratio of the attack when we overflow and pivot the stack into our spraying data.

OpenBSD PHK malloc is employed in the *mmap* heap spraying attack. We spray the heap directly by allocating large chunk, which is performed by controlling the *client_body_buffer_size* to be a size large enough, and the payload will be mapped in a continuous memory space by *mmap*.

Triggering *brk* syscall for large chunk allocation in glibc ptmalloc is a lot trickier in that 65536 large chunks should be allocated first before the *brk* enters. To overcome this, we can manually set the *n_mmaps_max* by intentionally invoking a *mallopt* routine, which is used to tune the ptmalloc parameters, to set the value to a smaller one, or just simply mimic a large chunk allocation by allocating a chunk slightly smaller than the threshold, which will thus be placed with *brk* syscall. *brk* behavior also varies depending on whether PIE/PIC is enabled or not. When PIE/PIC is enabled, *brk* base address randomization is similar to *mmap*, which has 28 bits entropy. If not enabled, *brk* base address will start from a lower address which is more predictable and can be easily exhausted by heap spraying.

In the experiment, we find that extra objects will be allocated along with the large chunks we sprayed. However, this affects little to the effectiveness of heap spraying attack even though unexpected data is mixed with our attacking payload. We can set the spraying chunk size as large as possible, so that effects of these extra allocated pages can be negligible. In our experiment, the chunk size to spray is set to 128KB, while on average, we have found that two extra pages will be mapped along with every chunk we sprayed in the experiments. So, we will have a probability of only 1/16 that we will jump to an unexpected place, regardless of other entropy effects. This is also the case for guard pages. As long as the size of the spraying chunk is large enough, we can control the probability drop at a tolerable level.

The result shows that heap spray in native execution is quite reliable. We perform heap spray with 4096 connections and 128KB *client_body_buffer_size*, resulting in a total mem-

Table 1: Security Evaluation of RLCA

	with RLCA	without RLCA
Fengshui (PHK)	Failed	Stable
Fengshui (ptmalloc)	Failed	Stable
Spray mmap (PHK)	Failed	1/2048
Spray brk (ptmalloc)	Partial	Stable

ory allocation of 512MB. This can exhaust 17 bits entropy in *mmap* or *brk* attack. Since Linux ASLR applies 28 bits entropy on x64 machine, we will have to guess the rest 11 bits with a probability of 1/2048. For *mmap* attack under x64 machine, this entropy reduction may still not be able to get a reliable padding. *brk* (without PIE/PIC), however, locates in a 32 bit address space that is more predictable than *mmap*. We are able to exhaust all the possibilities of the *brk* region with 512MB heap spraying. It is, thus, can be reliably exploited.

On the other hand, RLCA mitigates the entropy reduction to some degree. The *mmap* location is randomly distributed among all the unmapped addresses and is not placed continuously. The entropy reduced is limited, and it's impossible to guess a suitable address to land. Though, RLCA provides a more randomized *brk* memory region than the native one, the entropy reduction is still unavoidable. One solution is to set a smaller upper bound to the *brk* region size, and return failed if the requested size is larger than the threshold. The glibc ptmalloc will fall to using *mmap* when *brk* fails. We can thus mitigate the entropy reduction by tuning the threshold to *brk* region size.

4.2.3 Brute Force Enumeration Attack

Since Nginx forks out worker process to handle the incoming requests, every worker process share the same memory space as the parent process. After every crash of the worker process, it will re-spawn with the same memory layout as the initial state of the origin worker process. They have the same *brk* base address, the same *mmap* starting address, and the same subsequent *mmap*ed memory layout if given the same input. We can thus perform a brute force enumeration attack to search the memory exhaustively until we find the right place. This kind of attack can be used as a complement to heap spray attack to exhaustively guess the location of the payload. Entropy is no longer a critical consideration. We will send less data in spraying the payload, but, however, we will have to perform more tries to find our payload.

In heap spray attack, brute force enumeration may be needed to improve the reliability. Since spraying *mmap* still leaves 11 bits entropy, at most 2048 tries (1024 on average) should be made to perform a successful attack. For *brk* spraying attack, we can also brute force guessing the address of *brk* region so that we could spray less payload into the memory.

RLCA mitigates this attack as well. RLCA randomizes *mmap* placement every time it is invoked. *mmap* will return different address in every run, so the memory layout will be unpredictable. In *brk* heap spray attack, the attacker is still able to allocate a large continuous memory region with attacker controlled data. However, since there is a possibility that *brk* region can move when the preallocated size is not enough, we can mitigate the brute force enumeration attack with randomly moved *brk* region.

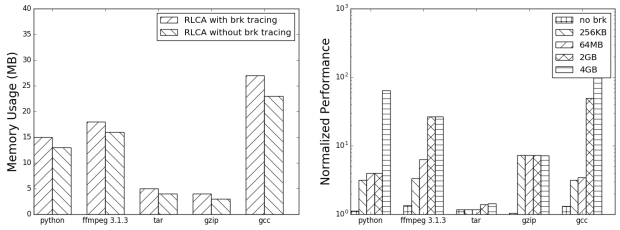


Figure 4: Brk Tracing Memory Overhead (Left) & Normalized Time For Different Upper Bounds (Right)

4.3 Performance Evaluation

4.3.1 Single Execution Performance

We evaluate the performance of RLCA against a list of softwares with RLCA protection, and comparing their results with native execution and nul-dynamoRIO results. Our testbed is one 6-Core Intel Xeon CPU E5-2643 v3, 32GB RAM.

The result is shown in Table 2. Comparing with nul-dynamoRIO results, RLCA presents a very small performance overhead and the memory overhead is negligible. We find that the performance and memory overhead introduced by dynamoRIO framework itself is significant. The initialization of the framework is slow and it casts more overhead than what it is introduced solely by RLCA. However, since the initialization performs only once through the whole lifetime of an application, we can thus ignore the performance overhead introduced by the DynamoRIO framework.

In the gcc experiment, the result is significantly poorer than any other experiment result. This is because gcc, unlike other testing programs, invokes additional subprocesses (e.g. as, collect2, ld) with execve syscall. This will trigger the dynamoRIO framework to reload and re-initialize every time execve is invoked. Extra overhead is introduced through this initialization process.

We have also shown the number of *mmap* syscall against the total malloc function called throughout the life-time of the application. Gzip and ffmpeg use internal memory allocators so we cannot get the exact statistic of malloc calls. We can deduce the number of large chunk allocations from the *mmap* syscall. On the other hand, RLCA does randomize every *mmap* encountered, including the large chunk allocations. On average, we have randomized approximately 12% of the memory allocations, with a performance overhead less than 20%, and memory overhead of less than 2%.

4.3.2 Brk Tracing Evaluation

To evaluate the performance of *brk* tracking, we run RLCA with different *brk* tracing upper bound against the same list of applications in the previous section. As is shown in Figure 4, RLCA with *brk* tracing takes significantly longer time, but in a moderate memory usage increase. The memory

Table 2: Performance & Memory Usage Comparison of native execution, NUL-dynamoRIO execution, RLCA

	python	ffmpeg 3.1.3	tar	gzip	gcc
native	0.016s/6MB	0.013s/10MB	0.042s/2MB	0.149s/2MB	0.045s/9MB
nul-dynamoRIO	0.26s/13MB	0.129s/16MB	0.110s/4MB	0.184s/3MB	0.753s/22MB
RLCA	0.29s/13MB	0.175s/16MB	0.129s/4MB	0.192s/3MB	0.985s/23MB
mmap vs malloc	76/2851	103/6	36/145	7/0	132/1544

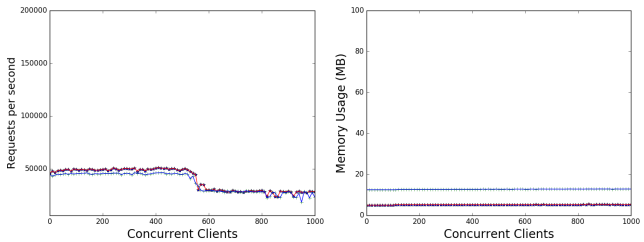
overhead is modest, of around 20%, considering the taint table and threadlocal memory space added. The performance overhead is relative high, however. We conclude the cause of such overhead as both taint tracing instrumentation and taint set operations. *brk* tracing requires to instrument after every instruction to store the potential taint value to the threadlocal storage. In DynamoRIO, additional calls will be instrumented to perform taint checking. The frequent calls introduce overhead to the overall performance. In tainting operations, every potential taint value is checked against the taint set for the further operations. This also introduces a considerable overhead.

As we have mentioned, we could tune the performance by limiting the upper bound of *brk* memory space size. In the experiments, we show that by tuning the upper bound, the performance can be optimized with a controllable taint table, while the tradeoff is that the transparency is hampered, since RLCA deliberately reduced the *brk* size. An upper bound of such would not cause any unstability because *brk* is originally designed to have a certain limit, and most of the softwares will have a failsafe measurement for such condition.

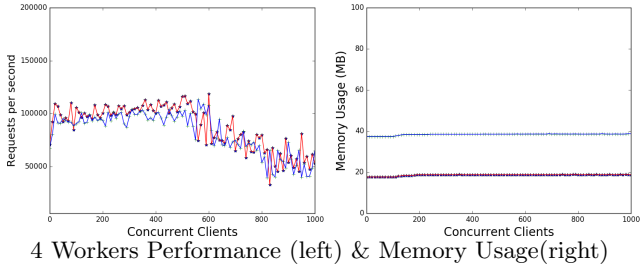
In the experiments, we have found that *brk* displacement is only triggered less than three times on every run. So, in normal programs, setting an upper bound of 64MB for *brk* would be enough, and we will have an overhead of only 3 times on average. We consider this as an acceptable overhead since *brk* will only be used for large chunk allocations in very extreme conditions.

4.3.3 Runtime Performance

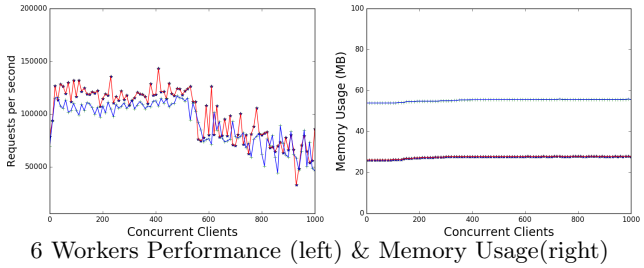
To evaluate the runtime performance of RLCA in real life. We run the test against Nginx with ABC benchmark [4] and evaluate the performance of RLCA under single worker, 4 workers and 6 workers. We run the tests from 1 to 1000 concurrent clients requesting web pages from our Nginx server. Each sending 100000 requests, and evaluate the requests processed per second and the average memory usage. The average performance overhead is 7.12% for single worker, 5.93% for 4 workers, 8.89% for 6 workers. The performance is more volatile for multi-worker situations because of the process scheduling. The memory usage has an overhead around 1 time the native. The results are shown in Figure 5.



Single Worker Performance (left) & Memory Usage(right)



4 Workers Performance (left) & Memory Usage(right)



6 Workers Performance (left) & Memory Usage(right)

Figure 5: Comparison of Performance & Memory Usage for Nginx (Red for native execution; Blue for RLCA)

5. DISCUSSION

We implement RLCA with the same *mmap/munmap/mremap* behavior as the underlying Linux kernel. If the memory requested by *mmap* with *MAP_FIXED* flag overlaps an already mapped memory, the memory region will be overwritten with the new memory. This may cause some crashes when it overlaps certain randomized data. It is possible that we can flag an exception in such condition, but we keep it as a Linux feature to make RLCA completely transparent to both application and kernel. We put our trust in the application developers that they know exactly what they are doing to map at a fixed address. One example is the loading of dynamic libraries. Glibc loader (ld.so) will *mmap* a large space at the first place, and then maps the data segment from the image file part by part with *MAP_FIXED* to override the previously mapped address.

Stack grows automatically when stack usage exceeds the allocated region. This may collide with the memory space allocated randomly by RLCA in some occasions. It would not be a problem in most cases if application developers are careful enough in stack usage management, because randomly placed memory would leave enough room for stack growth in most cases. However, to handle this issue, one possible approach is to keep track of all the stack allocations, and leave enough room above the stack to be un-mappable. Randomized allocations will not be placed too close above the stack, so the collision of stack growth can be mitigated. To identify the stack memory region, two situations should be considered separately, main process stack, which is allocated by *execve()* syscall, and thread stack, which is allocated with *mmap()* syscall. In the first scenario, the initial stack range can be determined by the RSP value. For the

latter case, thread stack can be determined by *MAP_STACK* flag in *mmap* syscall.

In the experiment we tested, we find that x64 memory space is large enough to tolerate the collisions we mentioned above, as well as a special thanks to the developers that write the excellent code.

Since RLCA provides a randomized *brk* solution, we can also protect small chunk in the *brk* memory space from heap spraying attack. We can set a threshold to force *brk* region to move periodically if the cumulative size requested through *brk* reaches this threshold, so that we can ensure that the address of *brk* memory space will be renewed from time to time. The attacker can never guess the address of the payload she sprayed with small chunks in the *brk* memory space.

6. RELATED WORK

PartitionAlloc [7] is a memory allocator designed and implemented in the chrome project. It allocates large chunk over about 1MB through *mmap*, but in each with a random address by utilizing the hint feature of *mmap* syscall. It is an application-specific memory allocator, however, that only applies to the chrome browser, and still it cannot mitigate the existing problems in the implementation of other memory allocators.

Works in ASLR and Address Space Re-randomization also relate to the *mmap/brk* randomization. Most of the works aim to provide more randomization to code base in order to mitigate code reuse attack. Only a few works have touched *mmap/brk* data region randomization. However, most of them require kernel modification.

PaX RANDMMAP [8] provides a Linux kernel patch to randomize the *mmap* and *brk* address. Bits from 12th to 27th are randomized on every *mmap* and the base address of *brk*. There is, however, research [29] shows that it is still buggy in some cases. On the Family 15h of AMD Bulldozer processors the randomization entropy can be reduced by 3 for instance.

ASLP [24] provides a comprehensive system for address space randomization on x86 machine through the whole lifecycle of a program, including an ELF rewriting tool to randomize ELF segments and functions, and a custom kernel to randomize user stack, *brk* and *mmap* addresses. ASLP randomizes the start address of *brk* and add a random offset between 0 - 4KB to provide sub-page randomization. *mmap* address is randomly selected from the 3GB x86 user space memory. An exception is generated when requesting *MAP_FIXED* address overlaps a mapped address.

RUNTIMEASLR [28] presents a re-randomization strategy to mitigate clone-probing attacks. The rational behind is to re-randomize the process address space at every fork(), so the attacker cannot figure out the memory layout by endless probing. RUNTIMEASLR only re-randomizes the memory layout on fork(), and tracks *mmap* only for code pointer fixing.

User-land ASLR solutions focus mostly on randomizing the memory layout at program load time. Heap address randomization is simply implemented by padding a random size in *brk* region and around chunks allocated.

TRR [41] provides a custom program loader to achieve this. Heap base is relocated randomly by growing the heap base with a random amount of space using the *brk()* system call. TRR only randomizes the memory layout at program load time. Runtime *mmap* randomization is not supported.

Address Obfuscation [13] is a x86 solution based on binary rewriting. Native codes are inserted directly into the binary image providing randomizations both in program load time and at runtime. *mmap* syscall is instrumented only in dynamic linker to provide randomization to dynamic load library image base address. Heap base address is padded with a random size on initial making the base address of heap unpredictable. Malloc is intercepted with a wrapper function to provide a random padding of 0 - 25% of size requested.

S Bhatkar et al. presents another load-time ASR solution [14] by designing a C compiler to add custom randomization code before the program started. Finer grained code randomization is provided. However, not much has been changed in heap randomization. *brk* base address randomization and chunk allocation randomization are still implemented by random padding.

7. CONCLUSION

In this paper, we revisit the heap security and assess the effectiveness of security memory allocators in large chunk protection. We find that nearly all the memory allocators fail to properly randomize the large chunk. Memory allocators put too much trust in the underlying system to provide a properly randomized large chunk placement through *mmap* syscall. In the Linux distributions, the *mmap* syscall is only randomized in base address, rather than the relative offset. We show that it is possible to attack memory allocator by large chunk fengshui and heap spray. To mitigate this problem, we present RLCA as a complement to security memory allocators to provide a runtime randomization for large chunk allocations. RLCA forces randomization in *mmap* and *brk* syscall by an efficient binary instrumentation. We show that for applications with RLCA, heap fengshui and heap spray attacks are successfully mitigated. And, the average overhead is less than 20%, while the runtime overhead is less than 10%.

All the code of RLCA and exploit are open-sourced at https://github.com/HighW4y2H3ll/RLCA.

8. REFERENCES

[1] Advanced heap spraying techniques. https://www.owasp.org/images/0/01/OWASL_IL_2010_Jan_-_Moshe_Ben_Abu_-_Advanced_Heapspray.pdf.

[2] Blind return oriented programming (brop). http://www.scs.stanford.edu/brop/.

[3] Cve-2014-0133. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0133.

[4] G-wan apachebench / weighttp / httperf wrapper. http://gwan.com/source/ab.c.

[5] Guard pages. https://www.us-cert.gov/bsi/articles/knowledge/coding-practices/guard-pages.

[6] Malloc des-maleficarum. http://phrack.org/issues/66/10.html.

[7] Partitionalloc. https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h.

[8] Pax randmmap. https://pax.grsecurity.net/docs/randmmap.txt.

[9] Php zend allocator. https://github.com/php/php-src/blob/master/Zend/zend_alloc.c.

[10] Understanding glibc malloc.

https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/.

[11] E. D. Berger. Heapshield: Library-based heap overflow protection for free. *UMass CS TR*, pages 06–28, 2006.

[12] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Usenix Security*, volume 3, pages 105–120, 2003.

[14] S. Bhatkar, D. C. DuVarney, and R. Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *Usenix Security*, 2005.

[15] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[16] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[17] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[18] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144, 2012.

[19] S. Cristalli, M. Pagnozzi, M. Graziano, A. Lanzi, and D. Balzarotti. Micro-virtualization memory tracing to detect and prevent spraying attacks.

[20] J. N. Ferguson. Understanding the heap by breaking it. *Black Hat USA*, pages 1–39, 2007.

[21] W. Gloger. Ptmalloc. *Consulté sur http://www. malloc. de/en*, 2006.

[22] D. R. Hanson. A portable storage management system for the icon programming language. *Softw., Pract. Exper.*, 10(6):489–500, 1980.

[23] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan. Preventing overflow attacks by memory randomization. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 339–347. IEEE, 2010.

[24] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC*, volume 6, pages 339–348, 2006.

[25] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook*. Wiley Indianapolis, 2004.

[26] A. Krennmair. Contrapolice: a libc extension for protecting applications from heap-smashing attacks, 2003.

[27] D. Lea. Dlmalloc, 2010.

[28] K. Lu, S. Nürnberger, M. Backes, and W. Lee. How to make aslr win the clone wars: Runtime re-randomization. 2016.

[29] H. Marco-Gisbert and I. Ripoll-Ripoll. Exploiting linux and pax aslr's weaknesses on 32-and 64-bit systems. 2016.

[30] O. Moerbeek. A new malloc (3) for openbsd. In *Proceedings of the 2009 European BSD Conference, EuroBSDCon*, volume 9, 2009.

[31] N. Nikiforakis, F. Piessens, and W. Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2013.

[32] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[33] W. Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[34] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 4, pages 159–169, 2004.

[35] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pages 45–54. IEEE, 2002.

[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[37] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[38] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[39] P. Team. Pax address space layout randomization (aslr). 2003.

[40] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *NDSS*, 2014.

[41] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 260–269. IEEE, 2003.

[42] Q. Zeng, M. Zhao, and P. Liu. Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 485–496. IEEE, 2015.

APPENDIX

A. CVE-2014-0133 EXPLOITING DETAIL

The vulnerability we exploit lies in the experimental SPDY protocol implemented in Nginx server. Figure 6 is the exploiting path of CVE-2014-0133. Code reference can be found in Code 2. The vulnerability lies in the function *ngx_http_spdy_state_save* that it fails to check the remaining length of the input buffer. This results in a heap based buffer overflow on the next *recv*.

For the very beginning, *ngx_http_spdy_init* will initialize the connection parameters, including handler callbacks, zlib related structures, etc, and allocates a one-page memory

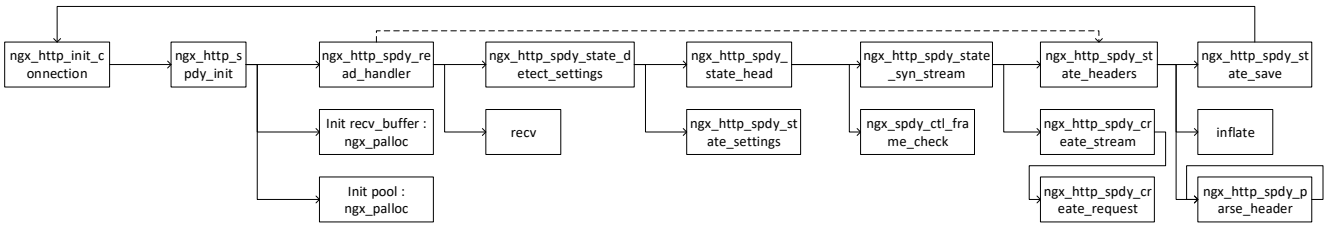


Figure 6: CVE-2014-0133 Nginx SPDY Heap Overflow Exploit Path

pool for allocating small objects. A receive buffer, *recv_buffer*, of size 256KB will be allocated in this initialization function to place the incoming SPDY data. This receive buffer is also the buffer we are going to overflow. *ngx_http_spdy_read_handler* performs the *recv* operations. It receives the incoming SPDY data into the *recv_buffer*. Nginx also stores a state buffer of size 16 bytes, used to store the remaining bytes which has not been able to be parsed from the previous SPDY request. On the initial *recv*, the counter of state buffer, *sc->buffer_used*, is null, and the state buffer data, *sc->buffer*, is meaningless. After the SPDY data is received into the *recv_buffer*, state buffer counter will be cleared, and the connection callback handler, *sc->handler*, will be called, which is previously registered as *ngx_http_spdy_state_detect_settings*. In *ngx_http_spdy_state_detect_settings* function, there are two paths leading to the vulnerable code. We can craft the payload to take the more straightforward path. At this stage, *ngx_http_spdy_state_head* is called to parse the header following the SPDY protocol, and dispatch the data to the corresponding routine. The routine leads to the vulnerable is *ngx_http_spdy_state_syn_stream*. This routine handles the incoming SPDY stream. *ngx_http_spdy_create_stream* and *ngx_http_create_request* are called in the routine to create a stream object. Another memory pool of one page size will be created in the *ngx_http_create_request*, and a series of small chunks will be allocated from the previous memory pool, including a 1KB header buffer to store unpacked SPDY stream header data. A subsequent call to *ngx_http_spdy_state_headers* will unpack the SPDY stream request into the header buffer, and parse the header.

There are three more constraints we have to bypass to reach the vulnerable code in *ngx_http_spdy_state_headers*. 1) The zlib inflation should return *Z_OK*. And, the length of input data, in *recv_buffer*, should be as large as possible (maxium 256KB - 32 bytes for state buffer) for the overflow. 2) SPDY frame headers, *sc->headers*, should not be null. *ngx_http_spdy_parse_header* should return *NGX_OK* or *NGX_DONE* to break the loop. 3) *buf->pos == buf->last*

The first constraint relates to the zlib inflation operation. The input buffer points to the compressed data location in *recv_buffer*, and the out buffer points to a temp buffer of size 1KB. The only condition to accommodate this constraint is to make the header larger than 1023 bytes, since the out buffer is set to take only 1023 bytes. And then, append our junk data to the compressed header so that our payload can fill the whole *recv_buffer* (minus 32 bytes). Since the out buffer is not large enough to hold all the header data, the inflation routine will return *Z_OK* for an authentic packed header.

For the rest two constraints, header parsing is the problem we have to take into consideration. There lies a loop in the vulnerable path to parse the unpacked SPDY frame header. We need to break the loop without exiting the routine. The only option left for us is to let the *ngx_http_spdy_parse_header* routine return *NGX_OK* or *NGX_DONE* with our crafted frame header. And, the third restriction indicates that the whole decompressed header can be completely parsed by the routine. To bypass this restriction, we construct the frame header following the routine in a key-value style with size exactly 1023 bytes, and compress the frame header data with a padding makes the payload larger than 1023 bytes.

Finally, we reach the *ngx_http_spdy_state_save* routine, where the vulnerability lies. In CVE-2014-0133, this routine does not check the length of the remaining data in *recv_buffer*, and stores it in *buffer_used*. Only 16 bytes of the remaining data are copied into the state buffer.

On the second *recv* in *ngx_http_spdy_read_handler*, the length to receive is unchanged, but the start pointer of the *recv_buffer* is incremented by the size we set previously in *buffer_used*, which is only several bytes less than 256KB. This is where the overflow takes place. In fact, since the callback handle is set to *ngx_http_spdy_state_headers* in *ngx_http_spdy_state_save*, we can even overflow this buffer multiple times with the same vulnerability in the subsequent series of recvs.

Nginx allocates objects using its internal pool-based memory allocator. If the requested size is smaller than the page size, the object will be allocated from the pool. Otherwise, it will use the underlying system malloc to perform the allocation. If the remaining size in the pool is not large enough to hold the object, a new page will be allocated to place the new object. The pool pages are also allocated by the system malloc.

Since *recv_buffer* will allocate only once on every worker, we will (sometimes have to) allocate chunks with comparative sizes to massage the memory space. The routine we exploit is the nginx fastcgi handler. For every http connection, the fastcgi handler will allocate a buffer of size, *client_body_buffer_size*. Such size of the buffer can be set in nginx config file as a configurable parameter.