MIRAGE : Randomizing Large Chunk Allocation Via Dynamic Binary Instrumentation

Zhenghao Hu
Shanghai Jiao Tong University
tonyhu@sjtu.edu.cn

Yuanyuan Zhang
Shanghai Jiao Tong University
yyjess@sjtu.edu.cn

Hui Wang
Shanghai Jiao Tong University
tony-wh@sjtu.edu.cn

Juanru Li
Shanghai Jiao Tong University
romangol@securitygossip.com

Wenbo Yang
Shanghai Jiao Tong University
wbyang@securitygossip.com

Dawu Gu
Shanghai Jiao Tong University
dwgu@sjtu.edu.cn

***Abstract*—Heap security relies heavily on the randomness of chunk allocations in memory allocators to mitigate heap fengshui and heap spraying attacks, 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 spraying 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 attacks under Nginx. We then present MIRAGE, 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 MIRAGE with a fine-grained randomization in *mmap*/*brk* syscall. And, MIRAGE imposes an acceptable overhead in performance.

I. Introduction

The war in heap security has never stopped. Among all the heap attacks proposed in the last few decades, the emerging of heap spraying [1] and heap fengshui [2] has proven to be the most effective and widely adopted methods to circumvent security mitigations. The idea of heap spraying is to exhaust as much memory space as possible in purpose of an ASLR [3] [4] 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 is able to 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.

In most cases, a finer-grained chunk randomization is very effective against heap fengshui and heap spraying. Security memory allocators [5] [6] are proposed to facilitate a more randomized chunk allocation in a design of BiBOP [7] style, with "Big Bag of Pages" acting as a memory pool. However, for all the security memory allocators, the randomization is partial. Chunks are only randomized within a manageable size, which is usually one page, since the randomize-able chunks are all organized into single pages. For a chunk of size over one page, a *mmap* syscall will be invoked directly, and the allocation of such a 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 spraying.

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 a 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 [5] and DieHarder [6] . Other general-purpose memory allocators have more diversed definitions. For instance, glibc ptmalloc [8] draws the line at 128KB, while dlmalloc [9] 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 [10] 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 [11] , 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 [12] 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 spraying attack by

978-1-5090-5569-2/17/$31.00

a limited amount, but is still far from a sound mitigation. We will show in Section IV-B2 that it is trivial to bypass guard pages in a heap spraying attack. Graffiti [13] provides an empirical solution to detect heap spraying attacks, but suffers from false positives and is limited to Intel CPUs with certain virtualization feature. PaX RANDMMAP [14] and ASLP [15] randomizes every *mmap* allocation so as to mitigate both heap fengshui and heap spraying, but, on the other hand, is 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 V). Still other fine-grained ASLR solutions [16] [17] [18] 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 spraying, either. Random padding can be easily bypassed with a large heap overflow in a heap fengshui attack, and the padding can also be negligible in a heap spraying attack as long as the chunk sprayed is large enough.

To overcome the problems of the previous works, we present MIRAGE to mitigate both large-chunk-based heap fengshui and heap spraying attacks by randomizing large chunk allocations at runtime. We design MIRAGE as a transparent layer between the system and the memory allocator to ensure large chunk randomization regardless of the types of the memory allocator. 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. *mmap* and *brk* syscalls are intercepted and handled exclusively to provide more randomization, but with strictly the same interface to both upper layer applications and underlying system.

In the evaluation, we designed large chunk fengshui and large chunk spraying experiment separately based on CVE-2014-0133 or CVE-2013-2028, under glibc ptmalloc and OpenBSD PHK malloc. Our result shows that MIRAGE can successfully mitigate both large-chunk-based attacks with a fine-grained randomization strategy. MIRAGE also introduces little performance penalty. The runtime performance overhead of MIRAGE is only 5 ˜10% on Nginx.

II. Large-Chunk-Based Attacks

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

1) 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.
2) The attacker can attack the application through heap spraying 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 are unable to corrupt or hijack the program, at least one vulnerability is required to trigger the attack.
3) There is no information leak in the target program or system, because if there is any, heap spraying 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 spraying, 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.

A. 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 [8] , OpenBSD PHK malloc [5] , and DieHarder [6] . 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.

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 spraying attacks are trivial. Since both *mmap* and *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

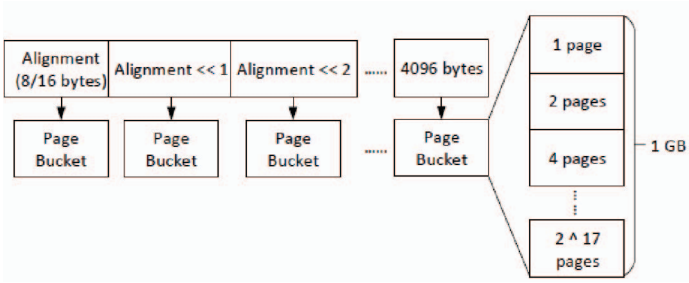


Fig. 1: DieHarder Internal Memory Structure

techniques can also be found both in publications and wild. [19] [20] [21]

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 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.

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 spraying, 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.

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-two. (Figure 1) Every chunk of the 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.

III. Design & Implementation

We design MIRAGE to protect large chunk allocations for both general purpose memory allocators and application specific ones. MIRAGE provides randomization for large chunks at runtime. In the design of MIRAGE, we aim to satisfy the following requirements:

- *Portability* : MIRAGE should be portable and easy to deploy. And, MIRAGE will not ask for any system modification to work correctly.
- *Comprehensiveness* : MIRAGE should be able to intercept all the *mmap* and *brk* syscalls through the entire life-cycle of the application.
- *Transparency* : MIRAGE should be memory allocator and operating system agnostic. The functionalities and features of the underlying system should be perserved.
- *Fine Grained Randomization* : MIRAGE should provide a fine grained randomization at runtime that spread through the entire virtual memory address space.
- *Acceptable Overhead* : Overhead introduced by MIRAGE should be acceptable.

We build MIRAGE as a client library on top of DynamoRIO 6.2.0 [22] to satisfy the above requirements. DynamoRIO provides runtime binary translation and instrumentation so that we can intercept any syscall regardless the underlying system or the upper-layer memory allocator. For *brk* migration, taint

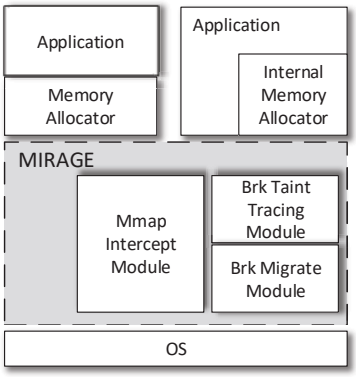


Fig. 2: MIRAGE Architecture Overview

tracking will also need dynamic instrumentation to record *brk* memory references. The overhead of DynamoRIO is substantial, but most of the overhead introduced is at program initialization stage - the runtime overhead is relatively small. For runtime services, like Nginx, the overhead of MIRAGE is only 5 ˜10%.

The design overview of MIRAGE is shown in Figure 2. MIRAGE is composed of two modules that separately handle *mmap* and *brk* syscall because of their different behavior. All the code of MIRAGE and exploit are open-sourced at https://github.com/HighW4y2H3ll/RLCA.

A. mmap module

mmap module intercepts all the *mmap/munmap/mremap* syscall, and redirect the memory mapping to a random unused address. The process memory layout is obtained from /proc/*pid*/maps file, and is stored in data structures in avoidance to repeatedly read and parse the maps file every time we perform an allocation.

For multi-page allocations, we first filter out all the memory holes large enough to hold the allocation. We then randomly select one from these candidates, and place the new allocation randomly in the hole. *MAP_FIXED* flag is preserved and it will force mapping a memory regardless the overlap, like what is done in Linux kernel. For *munmap*, if any part of the requested memory overlaps with an unmapped address, the un-mapping procedure should fail. And for *mremap*, if *MREMAP_FIXED* and *MREMAP_MAYMOVE* are both set, we should move the old address to a fixed new address regardless the overlap. If the *MREMAP_MAYMOVE* is solely set, we will unmap the old address, and map a new memory space randomly. And, 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.

Randomizing large chunks of relatively small size, e.g. one 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. We maintain a pool for single page allocations. When the page is requested, we will randomly select a page from pool. We guarantee the minimum randomness of page allocations by pre-filling the pool with memory pages to an adjustable threshold before the allocation. The filling process starts from the least sized holes in the memory layout, splitting each into single pages, and registers each into the page pool.

B. brk module

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. MIRAGE 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. Migrating *brk* memory space requires to fix all the memory references pointing to the old space. In this design, MIRAGE implements a taint tracking engine to keep track of *brk* memory references when *brk* is first called.

The overhead of taint tracking grows significantly with the size of the *brk* memory space. We optimize this with an adjustable threshold for the *brk* memory, which avoids the size of the taint table becoming too large. When the requested size exceeds the threshold, MIRAGE will return as failed, as what the Linux kernel does.

IV. Evaluation

In this section, we evaluate both the security and performance of MIRAGE. Our testbed is x64 Ubuntu 14.04.4 LTS, Linux 3.13.0-89-generic, with one 6-Core Intel Xeon CPU E5-2643 v3, and 32GB RAM. The target application is Nginx 1.4.7. In the security evaluation, we removed CVE-2014-0133 [23] patch in large chunk fengshui attack. For the large chunk spraying attack, CVE-2013-2028 [24] patch is removed in the experiment. 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.

A. 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 on the stack.

Listing 1: 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;
```

	with MIRAGE	*without MIRAGE*
Overflow PHK	Failed	Stable
Overflow ptmalloc	Failed	Stable
Spray mmap (PHK)	Failed	1/2048
Spray brk (ptmalloc)	Partial	Stable

TABLE I: Security Evaluation of MIRAGE

```
 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);
```

CVE-2014-0133 is a heap-based overflow in the Nginx SPDY protocol. As shown in Listing 1, 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.

B. Security Evaluation

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

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 spraying attack.

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. 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 metadata. 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 MIRAGE 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 maliciously 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 MIRAGE 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.

2) Heap Spraying 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 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 [25] 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. We adopt the technique introduced by blind-ROP [26] to acquire the image base address - enumerating the return address byte by byte when the worker thread stops crashing, so that we are able to build the gadgets with hardcoded addresses even when PIE/PIC is enabled.

OpenBSD PHK malloc is tested in the *mmap* heap spraying attack. We spray the heap directly by allocating large chunks, 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 by the *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 the effects of these extra allocated pages can be negligible. In our experiment, the size is set to 128KB, while on average, we have found that two extra pages will be mapped along with every chunk we sprayed. So, we will have a probability of only 1/16 that we will jump to an unexpected place. 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 spraying in native execution is quite reliable. We perform heap spraying attack with 4096 connections and 128KB client body buffer size, resulting in a total memory 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, MIRAGE 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, MIRAGE 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.

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*ped 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 spraying attacks. Entropy is no longer a critical consideration. We can send less data in spraying the payload, but, we will have to perform more tries to find our payload.

In the heap spraying attack, brute force enumeration may be needed to improve the reliability. Since spraying with *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.

MIRAGE mitigates this attack as well. MIRAGE 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 spraying 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.

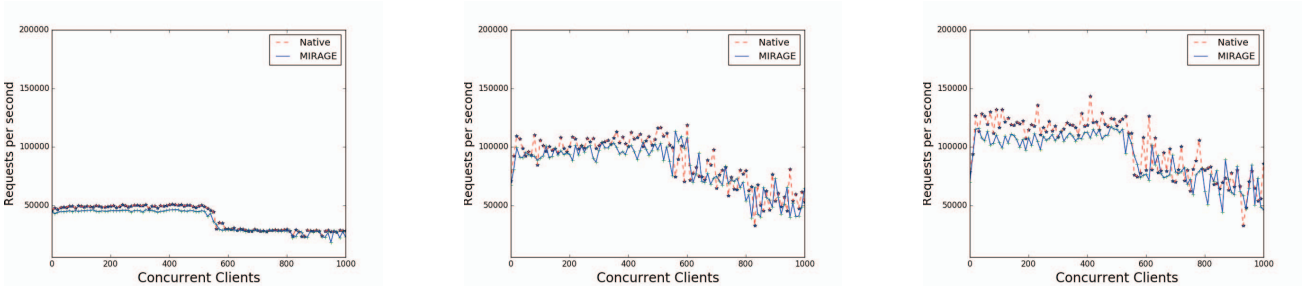
C. Performance Evaluation

To evaluate the runtime performance of MIRAGE in real life. We run the test against Nginx 1.4.7 under OpenBSD PHK malloc with ABC benchmark [27] and evaluate the performance of MIRAGE 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. 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.

V. Discussion

We implement MIRAGE 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 MIRAGE 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 in glibc. 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 MIRAGE 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, 2 situations should be considered separately: main process stack, which is allocated by execve() syscall, and thread stack, which is allocated with *mmap*()



Single Worker Four Workers Six Workers

Fig. 3: Comparison of Performance for Nginx

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 identifying MAP_STACK flag in *mmap* syscall. In the experiment we tested, 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 MIRAGE 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.

VI. Related Work

PartitionAlloc [28] 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. However, it is an application-specific memory allocator that only applies to the chrome browser, and still it cannot mitigate the existing problems in the implementation of other memory allocators.

TRR [18] provides a custom program loader as a user-land ASLR solution. 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.

Other works can be categorized as follows:

1) DBI Assisted: DrMemory [29], Memcheck [30] 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.

Iyer *et al.* [31] 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.

RUNTIMEASLR [32] presents a re-randomization strategy to mitigate clone-probing attacks. The rationale behind it 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.

2) Compiler/Kernel/Hardware Assisted: Graffiti [13] relies on EPT page-table, which is a virtualization feature of some Intel processors, to detect heap spraying attack. HeapSentry [33] requires to insert a kernel module to facilitate heap overflow protection. CRED [34], AddressSanitizer [35], HeapTherapy [36] 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.

PaX RANDMMAP [14] 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 [37] shows that it is still buggy in some cases. On the Family 15h of AMD Bulldozer processors the randomization entropy can be reduced by three for instance.

ASLP [15] provides a comprehensive system for address space randomization on x86 machine through the whole life-cycle 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.

Address Obfuscation [16] 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.

Bhatkar *et al.* presents another load time ASR solution [17] by designing a C compiler to add custom randomization code before the program started. However, *brk* base address randomization and chunk allocation randomization are still implemented based on random padding.

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 preferred 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 [38], or re-implements a much safer memory allocator [5] [6] [39] [40].

Wrapper approach is restricted to tracing and appending heap red-zones. 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.

VII. 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 common 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 spraying. To mitigate this problem, we present MIRAGE as a complement to security memory allocators to provide a runtime randomization for large chunk allocations. MIRAGE forces randomization in *mmap* and *brk* syscall by an efficient binary instrumentation. We show that for applications with MIRAGE, large chunk fengshui and spraying attacks are successfully mitigated. And, the runtime overhead is less than 10%.

References

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

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

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

[4] P. Team, "Pax address space layout randomization (aslr)," 2003.

[5] O. Moerbeek, "A new malloc (3) for openbsd," in *Proceedings of the 2009 European BSD Conference, EuroBSDCon*, vol. 9, 2009.

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

[7] D. R. Hanson, "A portable storage management system for the icon programming language," *Softw., Pract. Exper.*, vol. 10, no. 6, pp. 489–500, 1980.

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

[9] D. Lea, "Dlmalloc," 2010.

[10] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[11] "Php zend allocator," https://github.com/php/php-src/blob/master/Zend/zendalloc.c.

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

[13] S. Cristalli, M. Pagnozzi, M. Graziano, A. Lanzi, and D. Balzarotti, "Micro-virtualization memory tracing to detect and prevent spraying attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, pp. 431–446.

[14] "Pax randmmap," https://pax.grsecurity.net/docs/randmmap.txt.

[15] C. Kil, J. Jun, C. Bookholt, and J. Xu, "Address space layout permutation," *DSN 2006*, p. 194.

[16] 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*, vol. 3, 2003, pp. 105–120.

[17] ——, "Efficient techniques for comprehensive protection from memory error exploits." in *Usenix Security*, 2005.

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

[19] "Malloc des-male carum," http://phrack.org/issues/66/10.html.

[20] "Understanding glibc malloc," https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/.

[21] J. N. Ferguson, "Understanding the heap by breaking it," *black Hat USA*, pp. 1–39, 2007.

[22] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 133–144, 2012.

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

[24] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 227–242.

[25] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

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

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

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

[29] 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*. IEEE Computer Society, 2011, pp. 213–223.

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

[31] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, "Preventing overflow attacks by memory randomization," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 339–347.

[32] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make aslr win the clone wars: Runtime re-randomization," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2016.

[33] 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*. Springer, 2013, pp. 177–196.

[34] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector." in *NDSS*, vol. 2004, 2004, pp. 159–169.

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.
[36] Q. Zeng, M. Zhao, and P. Liu, "Heaptherapy: An efficient end-to-end solution against heap buffer overflows," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 485–496.
[37] H. Marco-Gisbert and I. Ripoll-Ripoll, "Exploiting linux and pax aslrs weaknesses on 32-and 64-bit systems," 2016.
[38] A. Krennmair, "Contrapolice: a libc extension for protecting applications from heap-smashing attacks," 2003.
[39] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Acm sigplan notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.
[40] E. D. Berger, "Heapshield: Library-based heap overflow protection for free," *UMass CS TR*, pp. 06–28, 2006.