From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel

Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang
Tianyi Xie, Yuanyuan Zhang[*] , Dawu Gu
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai, China

ABSTRACT

Since vulnerabilities in Linux kernel are on the increase, attackers have turned their interests into related exploitation techniques. However, compared with numerous researches on exploiting use-after-free vulnerabilities in the user applications, few efforts studied how to exploit use-after-free vulnerabilities in Linux kernel due to the difficulties that mainly come from the uncertainty of the kernel memory layout. Without specific information leakage, attackers could only conduct a blind memory overwriting strategy trying to corrupt the critical part of the kernel, for which the success rate is negligible.

In this work, we present a novel memory collision strategy to exploit the use-after-free vulnerabilities in Linux kernel reliably. The insight of our exploit strategy is that a probabilistic memory collision can be constructed according to the widely deployed kernel memory reuse mechanisms, which significantly increases the success rate of the attack. Based on this insight, we present two practical memory collision attacks: An object-based attack that leverages the memory recycling mechanism of the kernel allocator to achieve freed vulnerable object covering, and a physmap-based attack that takes advantage of the overlap between the physmap and the SLAB caches to achieve a more flexible memory manipulation. Our proposed attacks are universal for various Linux kernels of different architectures and could successfully exploit systems with use-after-free vulnerabilities in kernel. Particularly, we achieve privilege escalation on various popular Android devices (kernel version>=4.3) including those with 64-bit processors by exploiting the CVE-2015-3636 use-after-free vulnerability in Linux kernel. To our knowledge, this is the first generic kernel exploit for the latest version of Android. Finally, to defend this kind of memory collision, we propose two corresponding mitigation schemes.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Security.

Keywords: Memory Collision; User-after-free Vulnerability; Linux Kernel Exploit

1. INTRODUCTION

Recent mitigation efforts such as DEP, ASLR, stack canaries, and sandbox isolation significantly increase the cost to compromise user level applications and thus attackers turn their interests into OS kernels. Compared with user applications, current OS kernels are of high interest to attackers for two main reasons. First, attacking the core is always more effective. If the sandbox of a user application cannot be bypassed, attackers are not able to do anything that causes real damage to the system or other applications, especially on Android and iOS devices. However, if an attacker was able to compromise the kernel and achieve code execution in the kernel context, he could take the complete control of the system and there would be no need to bypass the sandbox. Second, less protection and mitigation schemes are applied to the OS kernel compared with that for the applications.

With the improvement of security in Linux kernel, vulnerabilities such as *logical errors* and *checks missing on arguments and privileges* become rare. It is also hard for attackers to exploit vulnerabilities like *NULL pointer dereferences* as well as *stack overflows* due to the stack canary applied in Linux kernel [7]. Thus the kernel heap related vulnerabilities such as *heap overflows* and *use-after-free* become significant targets for attackers to exploit nowadays. However, exploiting a heap vulnerability (e.g. use-after-free) in the kernel is still non-trivial although there are fewer related mitigation and protection schemes in action than with user applications. A particular heap layout is hard to be constructed in Linux kernel since various tasks are running in the kernel and have an impact on the kernel heap simultaneously. To effectively and efficiently exploit use-after-free vulnerabilities in Linux kernel, the following challenges need to be addressed:

Stability: As a modern operating system, Linux supports multi-thread scheduling, which means that large amounts of tasks may run on the system simultaneously. Every task has the possibility to influence the kernel, which may lead to the allocation or de-allocation of kernel objects at any time. The behavior of the kernel allocator is unstable and unpredictable under such circumstance, and this is the most troublesome part for the kernel exploit. A successful attack should overwrite the target kernel object without unexpected corruption.

[*] Corresponding author: yyjess@sjtu.edu.cn

CCS'15, October 12-16, 2015, Denver, CO, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813637.

Separation: Due to the inner workings of Linux kernel's allocators (SLAB and SLUB allocator, described in Section 3), different types of kernel objects cannot be stored in the same memory region. The memory overwriting based attack work needs to circumvent such restriction to fulfil the target.

Data Control: When memory gets overwritten, the new object appears at the location once occupied by the freed object and fill the freed object with its data. The covering data is crucial when exploiting a memory corruption bug like use-after-free. Thus it is important for the collision attack to not only occupy the space but also fully control the filling data.

To address these challenges, we propose a novel memory collision attack strategy to be universally applied to exploit use-after-free vulnerabilities in Linux kernel. The basic insight of our attack is that the memory allocation and reuse mechanisms of the kernel expose certain patterns, which can be leveraged to construct a memory collision (a probabilistic memory overwriting) with high success rate. For instance, due to physical memory limitation, the system always first recycles the recently freed memory for a future allocation, in order to improve performance and save energy. Such behavior reduces the entropy of the memory layout and leads to a high probability of a memory collision. Using this behavior in our attack allows us to exploit these *memory collision* style vulnerabilities with a high rate of appearance. That justifies why nowadays collision style vulnerabilities are likely to be occur and attackers always manage to exploit them.

As a proof of concept, two practical attacks are constructed based on our memory collision attack strategy:

Object-based attack: The first attack mainly uses the kernel buffers created by calling *kmalloc* to create a collision with the freed vulnerable object and fill it with the specific data. It is based on the observation that a successful memory collision in kernel can be achieved with the size separation provided by the SLUB allocator. Since objects of different types might share the same size, they can be arranged into one cache by the SLUB allocator with a well-designed attack. We present two types of object-based attacks. One is a collision attack between objects of the same size, which is stable but comes with many restrictions. The other is a collision attack between objects of different sizes, which circumvents the object isolation provided by the kernel allocator.

Physmap-based attack: The second attack mainly uses the physmap in the kernel to achieve memory collision and is more powerful. The physmap is a large virtual memory region inside the kernel address space that contains a direct mapping of all or a part of the physical memory. The usage of the physmap to bypass application level protections is first mentioned in [17]. And based on the previous work, we discover an unexpected and powerful use of the physmap, which is to re-fill the freed vulnerable objects. This leads to a generic, stable and reliable exploitation of use-after-free Linux kernel.

In short, our attack can completely bypass the separation provided by the SLAB/SLUB allocator. We find that almost all the use-after-free vulnerabilities in Linux kernel can be exploited by adopting our attack model. In fact, the proposed physmap-based attack overcomes most difficulties encountered while exploiting use-after-free bugs mentioned above, and is universally applied for both 32-bit and 64-bit versions of Linux on various architectures including the Android kernel. To validate the effectiveness and wide applicability of our attacks, a number of experiments are carried out. Particularly, We demonstrate a generic Android kernel exploit using **CVE-2015-3636**, a vulnerability credited to the author, to root most Android devices on the market nowadays including the 64-bit ones. Furthermore, two effective mitigation against the related attacks are also presented in this paper.

2. A BIRD'S-EYE VIEW

2.1 Use-after-free Vulnerabilities in Linux Kernel

When a use-after-free vulnerability is associated with a kernel object, the memory it once occupied can be accessed by attackers again after the de-allocation by kernel allocators. Considering the following code in Listing 1 as a typical example. It is a vulnerable kernel module that introduces a new *syscall* and one can allocate kernel objects of 512-byte in a cache by option 1 and free the objects previously allocated by option 2. When using option 3, a function pointer stored in a specified kernel object is to be invoked. If the object has been freed and then filled with data controlled by attacker, the **EIP**/**RIP** register for x86/x64 architecture or the **PC** register for ARM architecture is to be hijacked to injected shellcode and an arbitrary code execution in kernel context will be achieved.

Listing 1: Vulnerable Kernel Module

```
...
asmlinkage int sys_vuln (int opt, int index) {
    ...
    switch (opt) {
        case 1: // Allocate
            ...
            obj[total++] = kmem_cache_alloc(cachep,
                GFP_KERNEL);
            break;
        case 2: // Free
            ...
            free(obj[index]);
            ...
            break;
        case 3: // Use
            ...
            /* no status checking */
            void (*fp)(void) = (void (*)(void))(*(
                unsigned long *)obj[index]);
            fp();
            break;
    }
    ...
    /* Return index of the allocated object */
    return total - 1;
}

static int __init initmodule (void) {
    ...
    cachep = kmem_create_cache("vuln_cache", 512, 0,
        SLAB_HWCACHE_ALIGN, NULL);
    sct = (unsigned long **)SYS_CALL_TABLE;
    sct[NR_SYS_UNUSED] = sys_vuln;
    ...
}
...
```

In fact, it is non-trivial to precisely re-occupy the memory once belonged to the vulnerable object when exploiting such a kernel use-after-free vulnerability. The complexity and diversity of memory layout of Linux kernel makes it very difficult to arrange an exploit precisely. In contrast to user programs, to achieve memory overwriting in OS kernel is much harder. The hardness comes from the fact that many tasks are scheduled concurrently on one core and all of them may have impacts on the kernel heap. Thus attackers can no longer predict the precise memory layout of the kernel space. The working mechanisms of the kernel allocators may also sharply decrease the probability of a memory overwriting in the kernel, including the randomness of allocations, the separation of kernel objects of different types and the support of per-CPU caches. Without specific information leakage, attackers can only perform a **blind** memory filling trying to overwrite the critical memory region with slim probability of success.

2.2 Memory Collision Strategy

Our proposed attack introduces a novel memory collision strategy, which guarantees that the memory region controlled by attackers could overlap the critical memory region with significant stability. We illustrate in Figure 1 the principle of our proposed memory collision strategy. Basically, in order to achieve a memory collision with a vulnerable freed object, we expect the object to be allocated at the place where some other critical kernel objects controlled by us will be later placed. To reliably reach a memory collision, our attack leverages a set of characteristics of kernel operations (e.g., the style of object allocation). Since Linux kernel always **first** recycles freed memory for a future allocation due to physical memory limitation and for performance enhancement, our proposed attack strategy leverages this observation: Once an allocated vulnerable object is freed, the kernel will recycle the space occupied by that object for a recent allocation. The insight of our attack strategy is that we always try to find a **candidate** which is to be chosen by the kernel to reuse the freed memory once occupied by a vulnerable object. The candidate could be an object, a buffer or even a mapped area from the user space (*physmap*). Based on a thorough understanding of the kernel's allocation mechanism, we select reasonable candidates and intentionally arrange the attack by manipulating these candidates to turn a blind memory overwriting into a stable memory collision with high probability.

Specifically, in this paper we propose two concrete memory collision attacks with different attack surface respectively. The first attack constructs memory collision by following the working mechanism of kernel allocators and conducts an attack under several heap protections (e.g. the separation among objects of different types). The second attack relies on the fact that a vulnerable object can also collide with a mapped area in kernel memory, where the candidate is typically not a single kernel object. By utilizing mapped memory, the second attack becomes much more universal since it does not concern the separation of itself with other kernel objects. These two attacks can be widely applied for exploiting use-after-free vulnerabilities in Linux kernel and overcome most difficulties brought by kernel allocators. In the following, we detail the key part of these two attacks.

3. OBJECT-BASED ATTACK

In this section, we present the details of object-based memory collision attack in Linux kernel. Before discussing the attack, we first introduce the working mechanism of kernel allocators. Then we introduce two types of object-based memory collision attacks: the memory collision happens between objects of the same size or between objects of different sizes.

3.1 Memory Allocation of Linux Kernel

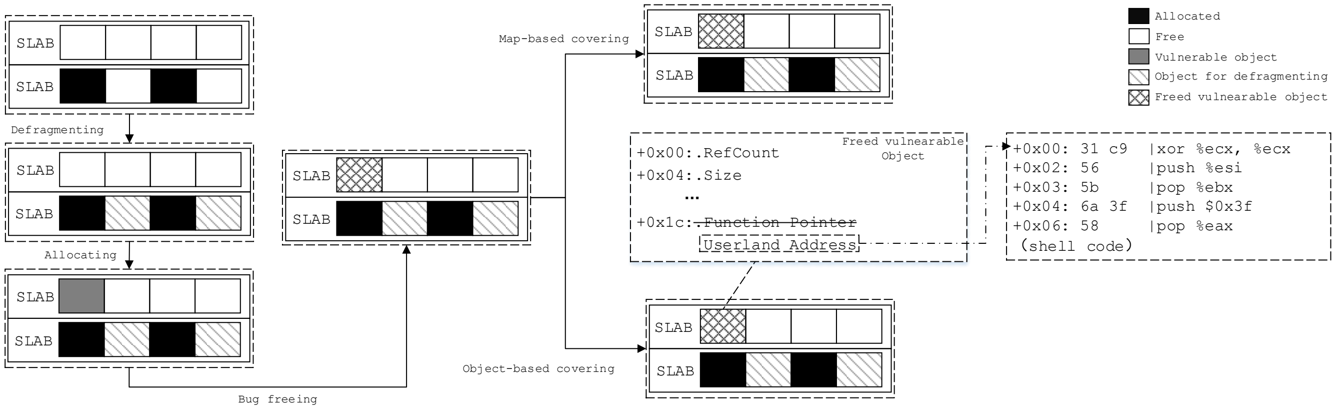
In Linux kernel, the SLAB/SLUB allocators are responsible for allocations of kernel objects. For kernel objects of a specific type, a corresponding storage unit is created by the SLAB allocator as a container, which is called *SLAB cache*. It contains the data associated to objects of the specific kind of the containing cache [18]. Moreover, there are generally two interfaces to allocate objects in Linux kernel. One is *kmem_cache_alloc* [25], for which a type of a SLAB cache should be specified; The other one is *kmalloc* [25], which only needs a allocation size without a cache type. The objects created by invoking *kmalloc* are still classified into different SLAB caches according to their sizes, These SLAB caches are named as *kmalloc-size* SLAB caches. Another important kernel allocator called SLUB allocator has been in use in Linux kernel in 2008 [10] and improves the performance of the SLAB allocator.

The SLAB/SLUB allocators introduce mainly two restrictions to an attack. First, the heap management mechanism adopted by Linux kernel generally prevents attackers from creating memory collisions between kernel objects. SLAB caches separate kernel objects of one type from those of another type. It is therefore impossible to insert a new object into the free position of a SLAB cache and let the objects of two different types to be stored in one cache at the same time. When a use-after-free vulnerability in Linux kernel is to be leveraged, such separation brings difficulties for attackers to create memory collisions between kernel objects of different types. Second, considering a typical state of the kernel heap, when an object is to be allocated, there might exist several half-full SLAB caches which are able to store it. The holes in these SLAB caches should be allocated by kernel allocators with higher priority. To ensure that vulnerable objects are allocated into recently created SLAB caches instead of existing ones, a reliable attack must consider this property and try to first fill every hole in these half-full SLAB caches. This process is often referred to as **defragmentation**.

3.2 Collision between Objects of the Same Size

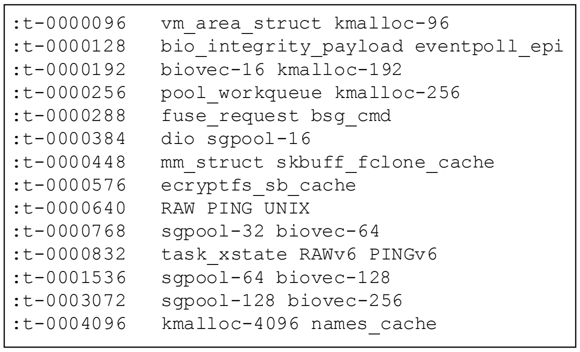
We first present a memory collision applied without breaking the size rule provided by the SLUB allocator.

Objects have diverse sizes among various Linux kernels due to different kernel sources and configurations during the compilation. Figure 2 illustrates a part of the result of executing *slabinfo* on a 32-bit Linux. In fact, the SLUB allocator tries to merge kernel objects of the same size instead of the same type into one cache, which helps to reduce overhead and increases cache hotness of kernel objects. As shown in Figure 2, the kernel objects of different types are classified into the same cache if they have an identical size. For example, both the *vm_area_struct* object and the *kmalloc*-96 object have a size of 96 bytes, which indicates that these two objects have a high opportunity to be allocated into the same cache in the kernel. This behavior



```
+0x00: 31 c9   |xor %ecx, %ecx
+0x02: 56      |push %esi
+0x03: 5b      |pop %ebx
+0x04: 6a 3f   |push $0x3f
+0x06: 58      |pop %eax
 (shell code)
```

Figure 1: Memory Collision Attack



```
:t-0000096   vm_area_struct kmalloc-96
:t-0000128   bio_integrity_payload eventpoll_epi
:t-0000192   biovec-16 kmalloc-192
:t-0000256   pool_workqueue kmalloc-256
:t-0000288   fuse_request bsg_cmd
:t-0000384   dio sgpool-16
:t-0000448   mm_struct skbuff_fclone_cache
:t-0000576   ecryptfs_sb_cache
:t-0000640   RAW PING UNIX
:t-0000768   sgpool-32 biovec-64
:t-0000832   task_xstate RAWv6 PINGv6
:t-0001536   sgpool-64 biovec-128
:t-0003072   sgpool-128 biovec-256
:t-0004096   kmalloc-4096 names_cache
```

Figure 2: Partial result of executing slabinfo on 32-bit Linux

allows attackers to create memory collisions between kernel objects of the same size.

The vulnerable module introduced in 2.1 is used to illustrate how our attack fill freed objects with controlled data. In that module, the size of every freed vulnerable object is 512 bytes. In order to introduce a collision, a candidate object is selected to re-occupy a previously freed space which has a different type but with the same size (512 bytes), based on the size rule of the SLUB allocator. Meanwhile, in order to control the content of freed space, the data of a proper candidate object should also be assigned by an attacker. Thus *kmalloc-size* buffer in kernel is the best choice due to its easy allocation, diverse sizes and capability of fully controlling the re-filling data. For example, a transfer buffer will be allocated by *kmalloc* during the process of *sendmmsg*. And attackers can set the buffer size as it represents the length of the control message and can also control the data in the buffer as it represents which control message one wants to deliver.

The following code in Listing 2 leverages the use-after-free vulnerability in the malicious kernel module in Section 2.1 to compromise the kernel and execute arbitrary kernel code by applying object-based collision attack. The most important part, *exploiting*, involves four essential steps including *allocating objects*, *freeing objects*, *overwriting freed objects* and *re-using freed objects*. Note that the length of the buffer is 512 bytes, which is equal to the size of the vulnerable object. And the parameters M and N can be specified by attackers based on the actual situation.

Listing 2: Object-based Attack

```
/* setting up shellcode */
void *shellcode = mmap(addr, size, PROT_READ |
     PROT_WRITE | PROT_EXEC, MAP_SHARED | MAP_FIXED
     | MAP_ANONYMOUS, -1, 0);
...

/* exploiting
 D: Number of objects for defragmentation
 M: Number of allocated vulnerable objects
 N: Number of candidates to overwrite
*/

/* Step 1: defragmenting and allocating objects */
for (int i = 0; i < D + M; i++)
    index = syscall(NR_SYS_UNUSED, 1, 0);
/* Step 2: freeing objects */
for (int i = 0; i < M; i++)
    syscall(NR_SYS_UNUSED, 2, i);
/* Step 3:  creating collisions */
char buf[512];
for (int i = 0; i < 512; i += 4)
    *(unsigned long *)(buf + i) = shellcode;
for (int i = 0; i < N; i++) {
    struct mmsghdr msgvec[1];
    msgvec[0].msg_hdr.msg_control = buf;
    msgvec[0].msg_hdr.msg_controllen = 512;
    ...
    syscall(__NR_sendmmsg, sockfd, msgvec, 1, 0);
}
/* Step 4: using freed objects (executing shellcode)
        */
for (int i = 0; i < M; i++)
    syscall(NR_SYS_UNUSED, 3, i);
```

3.3 Collision between Objects of Different Sizes

The attack described in Section 3.2 has an obvious weakness: it can only be applied when the size of a vulnerable object is aligned to one of possible *kmalloc* sizes, which are mainly powers of 2. Considering such situation that a vulnerable object has a size of 576 bytes, neither *kmalloc*-512 nor *kmalloc*-1024 objects are able to make collisions with the vulnerable object by the solution mentioned in

Section 3.2. Thus a more universal collision attack is required.

Assume that the vulnerable kernel module is modified on Line-27 and the size property of the cache is changed from 512 bytes to 576 bytes, we present an advanced attack which ignores the size of a target vulnerable object. And this time the attack still adopts kernel buffers of *kmalloc-size* type as candidates for memory collisions.

For the SLAB allocator in Linux kernel, if all objects in one SLAB cache are freed, the entire SLAB cache is going to be recycled for a future allocation. It reveals the fact that freed SLAB caches can be later used to hold objects of a completely different type, which allows to overcome size-isolated barriers. Thus, for our new attack, several new SLAB caches are created and filled with vulnerable objects in the very beginning. Note that all of the objects stored in these SLAB caches are the targets to collide with. Then by triggering the use-after-free vulnerability, all of these objects are released but still can be accessed by attackers. When all the objects in these SLAB caches are freed, the space of these SLAB caches previously created for vulnerable objects is going to be recycled by the kernel. And that freed space will be used later for *kmalloc-size* buffers created by invoking *sendmmsg*.

In fact, we can still use the same code listed in Section 3.2 to exploit the modified vulnerable kernel module and execute kernel shellcode. Either *kmalloc*-256 or *kmalloc*-128 buffers can be chosen as candidates to overwrite freed memory. And we also need larger M and N parameters to guarantee the reliability in this attack.

4. PHYSMAP-BASED ATTACK

The object-based attack by memory collisions between kernel objects of different sizes has an obvious weakness, the uncertainty. In this section, we present a more universal attack which leverages a specific mapped area called *physmap* in kernel memory without the weaknesses mentioned above. In fact, the physmap is originally used in Ret2dir technique [17] to bypass currently applied protections in Linux kernel. Because the data crafted by attackers in user space is directly mapped by the physmap into kernel space, thus the physmap can be used to rewrite the kernel memory previously occupied by freed vulnerable object and exploit use-after-free vulnerabilities.

Once attackers call *mmap* with an expected virtual address in user space and then call *mlock* on that virtual address, these pages in user space may be directly mapped into the physmap in kernel space. Therefore, the attack is performed by repeatedly invoking *mmap* in user space and spraying proper data in the physmap area. For the sake of convenience, the *physmap* mentioned in the rest of the paper represents the part of the directly mapped space in kernel which has already been filled with the payload sprayed by attackers.

Again, we illustrate our attack by exploiting the vulnerable kernel module mentioned in Section 2.1. As the current intended approach is to take advantage of the physmap to make a collision attack and rewrite freed objects, all the caches which contain target vulnerable objects should be recycled by the kernel for future allocation because the physmap never occupies the virtual memory in use. At the beginning of a physmap-based attack, defragmentation by certain kernel objects that have the same size of vulnerable objects is conducted (as mentioned in Section 3). They are used to pad free holes inside all half-full SLAB caches. And then new and clean SLAB caches will be created by kernel allocators when vulnerable objects are going to be allocated.

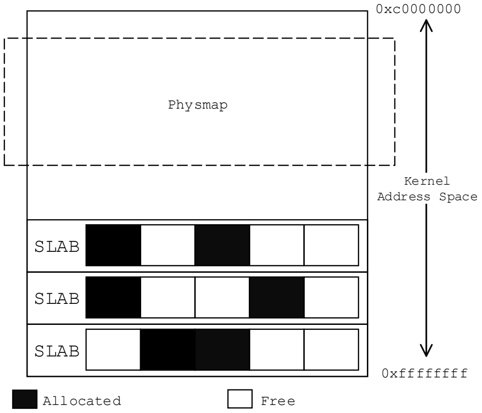


Figure 3: Kernel Memory Layout

Later, after all of these vulnerable objects are released by vulnerable syscalls, a certain amount of free SLAB caches once storing these vulnerable objects are generated and recycled by Linux kernel. They may be served as expanding area of the physmap where memory collisions happen in the future.

However, to improve the probability that memory collisions happen between target objects and the physmap, the location where kernel objects are allocated should be lift up. The layout of the kernel memory including the SLAB caches and the physmap is shown in Figure 3. It can be seen that the physmap begins at a relatively low virtual address, meanwhile SLAB caches usually gather at a higher address. Our goal is to create memory collisions between these two areas. Due to the capacity of physical memory, the total size of data sprayed in the physmap has an upper bound. Furthermore, the physmap always tends to require a piece of complete free memory of a certain size for expansion.

If vulnerable objects are allocated immediately after defragmentation, the physmap might not be able to expand and eventually cover the SLAB caches where once target objects are placed.

Thus our plan is to spray kernel objects in group. For each group, a certain amount of kernel objects are sprayed for *padding* and then several vulnerable kernel objects are to be allocated, which is treated as the target to be collided with. That makes these vulnerable objects dispersedly appear in kernel space, which sharply increase the probability of memory collisions between the physmap and vulnerable objects. In addition, if a target vulnerable object can be easily allocated and de-allocated by attackers in their programs, a proper choice of the padding object is just the vulnerable object itself.

Next, the objects for padding can be released through the intended way, while those vulnerable objects should be released by triggering a use-after-free vulnerability. After de-allocating all allocated objects, the memory previously occupied by these objects is released and small pieces of continuous freed memory might be merged into a larger piece, which is later provided for expansion of the physmap.

Note that the re-filling step should be conducted immediately to avoid freed memory to be corrupted by allocations of kernel from some other active tasks running on the system. And this time we use the physmap to introduce memory collisions with vulnerable kernel objects. In order to fill the physmap in kernel space with data completely user-controlled, *mmap* is repeatedly called with a large size as many times as possible. And for every piece of virtual memory returned back from *mmap*, we fill that memory with user-specified data and call *mlock* on it. As the physmap filled with proper payload grows, the freed memory previously occupied by vulnerable objects is eventually covered and memory collisions are successfully achieved.

The following code shows the *exploiting* part of the proposed physmap-based attack against the malicious kernel module.

Listing 3: Physmap-based Attack

```
/* exploiting
 D: Number of objects for defragmentation
 E: Iterations of object spraying
 P: Number of objects for padding in one group
 V: Number of allocated vulnerable objects in one group
*/

/* Step 1: defragmenting */
for (int i = 0; i < D; i++)
    syscall(NR_SYS_UNUSED, 1, 0);
/* Step 2: object spraying */
p = 0; v = 0;
for (int i = 0; i < E; i++) {
    for (int j = 0; j < P; j++)
        pad[p++] = syscall(NR_SYS_UNUSED, 1, 0);
    for (int j = 0; j < V; j++)
        vuln[v++] = syscall(NR_SYS_UNUSED, 1, 0);
}
/* Step 3: freeing */
for (int i = 0; i < p; i++)
    syscall(NR_SYS_UNUSED, 2, pad[i]);
for (int i = 0; i < v; i++)
    syscall(NR_SYS_UNUSED, 2, vuln[i]);
/* Step 4: creating collisions */
unsigned long base = 0x10000000;
while (base < SPRAY_RANGE) {
    unsigned long addr = (unsigned long)mmap((void *)base, 0x10000000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_SHARED | MAP_FIXED | MAP_ANONYMOUS, -1, 0);
    unsigned long i = addr;
    for (; i < addr + 0x10000000; i += 4) *(unsigned long *)(i) = shellcode;
    mlock((void *)base, 0x10000000);
    base += 0x10000000;
}
/* Step 5: using freed objects (executing shellcode) */
for (int i = 0; i < v; i++)
    syscall(NR_SYS_UNUSED, 3, vuln[i]);
```

Note that if values stored inside target vulnerable objects can be read out by specific syscalls, then an additional process in step 4 can be used to improve the efficiency and accuracy of physmap-based attack. First, besides some important values sprayed in the physmap to overwrite key entries at certain offset of freed vulnerable objects to avoid kernel crashes and execute kernel codes, we also spray a specific magic value like `0xdeadbeef` in the physmap. Then for every vulnerable object, one value inside it is read out at each stage after calling *mmap*. If it is directly equal to that magic value or correctly reflects the magic value once filled in, then a memory collision happens and the physmap spraying should be stopped.

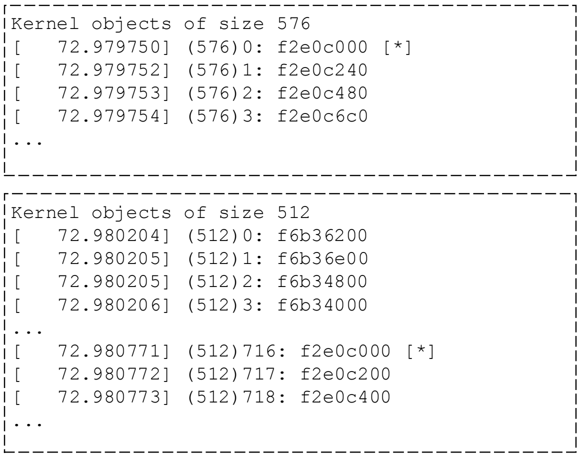
5. EFFECTIVENESS OF THE ATTACK

In this section, we evaluate the effectiveness of the proposed object-based memory collision attack and physmap-based memory collision attack.

5.1 Object-based Attack

5.1.1 Feasibility Analysis

For the attack based on memory collisions between the kernel objects, the generated message of a custom kernel module shown in Figure 4 illustrates the details. In the module a specific cache which holds objects of size 576 is created, then several kernel objects are allocated which belong to this cache and the virtual addresses of these objects are recorded. Then the kernel module frees all the objects in that cache and allocate 1024 buffers of size 512 by invoking *kmalloc* and their virtual address are also recorded. By viewing the kernel messages that the kernel module printed out, it can be seen that a memory collision happens when the 716th *kmalloc*-512 buffer is created because its virtual address is the same as the virtual address of the first object of size 576, which shows the feasibility of object-based attack.



```
Kernel objects of size 576
[   72.979750] (576)0: f2e0c000 [*]
[   72.979752] (576)1: f2e0c240
[   72.979753] (576)2: f2e0c480
[   72.979754] (576)3: f2e0c6c0
...

Kernel objects of size 512
[   72.980204] (512)0: f6b36200
[   72.980205] (512)1: f6b36e00
[   72.980205] (512)2: f6b34800
[   72.980206] (512)3: f6b34000
...
[   72.980771] (512)716: f2e0c000 [*]
[   72.980772] (512)717: f2e0c200
[   72.980773] (512)718: f2e0c400
...
```

Figure 4: Memory Collision between Kernel Objects

5.1.2 Advantages

The object-based attack described in Section 3 uses objects allocated by *kmalloc* to re-fill the memory of vulnerable freed objects. Since hundreds of flows in Linux kernel involve creating *kmalloc-size* buffers, objects of *kmalloc-size* type are the most controllable candidates to make memory collisions since it is easy for attackers to allocate in user programs by syscalls. For example, a buffer is created by *kmalloc* during the process of *sendmmsg* in kernel, which is used to hold control messages during message passing. And when one writes to one side of the pipe from the other side, *kmalloc-size* buffers are allocated to hold temporary data.

The second advantage of these *kmalloc-size* buffers is that both the content and the size are user-controlled. Since target vulnerable objects have different sizes in different cases, the size of a candidate object should be controlled to follow the size of a vulnerable object. For many kinds of *kmalloc-size* objects, they are often seen as buffers which store data from user space. That brings the full control of the overwriting content and thus attackers are able to set addresses of their injected codes or kernel ROP gadgets inside vulnerable objects.

5.1.3 Limitations

The object-based attack still faces some serious limitations in practice.

- For the attack based on collisions between kernel objects of the same size, the size of a vulnerable object has to be aligned to one possible *kmalloc* sizes, otherwise no valid *kmalloc* buffers can be found to cover that freed vulnerable object.
- For the attack based on collisions between kernel objects of different sizes, the main problem is its uncertainty. Without information leakage, definitely attackers do not clearly know whether target vulnerable objects have been covered by other objects of different sizes, the probability of a successful overwriting sharply decreases compared to the former object-based attack.

5.2 Physmap-based Attack

5.2.1 Feasibility Analysis

The feasibility of physmap-based attack depends on whether the physmap could eventually cover the SLAB caches where vulnerable objects store. It is discussed for two different platforms as follows:

32-bit Linux kernel: For 32-bit Linux platforms on most desktop PCs and Android devices nowadays, kernel space starts at `0xc0000000` and ends at `0xffffffff` as shown in Figure 3. As described in [17], the physmap starts at `0xc0000000` and is supposed to have a size of 891MB on x86 architecture and 760MB on ARM architecture, which indicates that the physmap ends at `0xf7b00000` on x86 and `0xef800000` on Android (ARM). Based on the fact that kernel objects start to be allocated at virtual addresses which are in range of `0xd0000000` ∼ `0xf0000000` in SLAB caches and after the object spraying step of physmap-based attack, the vulnerable objects are supposed to be uniformly allocated in kernel space. Thus when the physmap grows through spraying, it has large probability to cover target SLAB caches.

64-bit Linux kernel: For 64-bit Linux platforms, kernel space starts at `0xffff880000000000` on x86_64 architecture and for ARM architecture, it starts at `0xffffffc000000000`. As described in [17], the physmap starts at the location where kernel space starts and is supposed to have a size of 64TB on x86_64 and 256GB on Android (ARM). Considering the fact that 64-bit systems only use 48 bits for addressing, the range of the physmap covers the entire kernel space. Although allocations of kernel objects on 64-bit Linux platforms behave more random than they do on 32-bit Linux platforms, kernel objects are still placed in an area of certain size in kernel space when no memory pressure exists. For x86_64, kernel objects usually start to be allocated at virtual addresses which are in range of `0xffff880000000000` ∼ `0xffff8800ffffffff` and for ARM, kernel objects start to be allocated at virtual addresses which are in range of `0xffffffc000000000` ∼ `0xffffffc0ffffffff`. However, for any device or PC which has a 64-bit kernel inside, it has a relatively large RAM size which is commonly not less than 2GB. And after the object spraying step of our physmap-based attack, there is a substantial probability that memory collisions happens between the physmap and target SLAB caches.

Generally speaking, physmap-based attack is considered to be effective both for 32-bit kernel and 64-bit kernel. In fact, the size of the entire kernel space is not the only factor to successful collisions. The size of the RAM size also plays an important role. Low RAM size may hurt the effectiveness of physmap-based attack since it limits the total amount of data an attacker is able to spray in the physmap.

5.2.2 Advantages

Stability: The attack through the physmap is much more stable than object-based attack. In fact, the only thing requiring an attacker to do is to repeatedly map memory in user space and fill it with proper payload which can be easily operated in the user program. Overwriting through the physmap does not allocate any kernel object by kernel allocators and attackers do not need to be informed of kernel memory layout at any time.

Note that the physmap and all SLAB caches are stored in kernel space simultaneously, and surly they do not have their own private space due to limited virtual address space for Linux kernel. Considering the certain distance between initial locations of the SLAB caches storing vulnerable kernel objects and the physmap, the object spraying step makes the vulnerable objects not gather in one place but appear at different places in the kernel. That also ensures the stability of such a probabilistic attack strategy.

Finally, the additional step of physmap-based attack also improves stability. When attackers can be informed that vulnerable kernel objects have already been correctly overwritten then further kernel spraying which may bring unexpected results is no longer needed.

Separation: The physmap is like a nightmare since it circumvents the separation of kernel objects provided by kernel allocators not from the internal mechanism but from an overall view on the memory management of Linux kernel. The key point is that kernel needs to recycle freed memory for future use and do not have the ability to divide the whole kernel memory space into different parts for different uses due to limited memory size and the efficiency requirement of the kernel. That leaves the physmap, one of the best **candidates**, an opportunity to overwrite the freed memory once occupied by vulnerable objects and create stable memory collisions.

Note that physmap-based attack does not care about whatever the size of a vulnerable object is and the type of a vulnerable object is, which means a thorough bypassing of the separation protection provided by Linux kernel.

Data control: Since the data sprayed in the physmap all comes from the data filled in *mmaped* memory in user space, it is for sure fully user-controlled. Any malicious content attackers desire for is able to be filled into the freed memory once occupied by a target vulnerable object.

Wide applicable scenarios: Although the physmap-based attack is designed to achieve a generic approach to exploiting use-after-free vulnerabilities, it can be applied to exploit other types of vulnerabilities in Linux kernel based on the fact that the physmap has the ability to cover nearly any free space in kernel. One possible usage of physmap-based attack is to exploit uninitialized memory vulnerabilities. In fact, the physmap-based attack is definitely helpful and effective when a free space is needed to be occupied when exploiting different kinds of kernel vulnerabilities.

5.2.3 Comparison with Previous Techniques

Previous techniques to achieve overwriting on kernel objects which have use-after-free vulnerabilities always try to leverage the features of kernel allocators. For example in [6], attackers manage to overwrite a specific kernel object of size 224 with another kernel object of a close size, 256. For this case, the stable overwriting can also be achieved by physmap-based attack. However, previous techniques become useless in the following situations meanwhile physmap-based attack keeps performing well:

Various sizes: Previous techniques always try to place a different kernel object of similar size into the area once occupied by a vulnerable object to exploit kernel use-after-free vulnerabilities. However the size of a target vulnerable object always varies. Take the CVE-2015-3636 as an example (details are introduced in Section 6.2), vulnerable PING socket objects may have at least four different sizes on Android devices of different kinds of brands. And also the sizes of a vulnerable kernel object are not identical in 32-bit kernel and 64-bit kernel. Facing such situations, previous techniques have to hard-code in exploit codes to achieve stable overwriting for different kernels. Thus several different versions of the attack programs are required. If the size information cannot be known in advance, then previous techniques become much less effective since what types of kernel objects can be used to overwrite vulnerable ones can no longer be determined. By contrast, physmap-based attack is simple and intuitive. The only thing an attacker needs to do is to iteratively calling *mmap*. One generic exploit can be achieved by physmap-based attack.

Uncontrollable content: If a use-after-free vulnerability is desired to be exploited, not only a vulnerable object is needed to be overwritten but also the content of that object has to be under control. Previous techniques may have limited choices of kernel objects to make collisions and at many times these candidates may be the internal ones and attackers cannot set proper payload they want in freed memory, which makes an exploitation hard to complete afterwards. When it comes to physmap-based attack, all these memory used to overwrite vulnerable objects are generated by calling *mmap* in user programs and the content is naturally under control.

Multi-threading support: Previous techniques may need relatively accurate prediction on kernel heap layout. But when an attack program is executing meanwhile many other tasks scheduled on the core may influence memory layout of the kernel heap, which brings much uncertainty. This factor sharply decreases the success rate of kernel attacks based on previous techniques. By contrast, with the help of object spraying and physmap spraying in physmap-based attack, large amount of instances are created and almost all of the kernel space is occupied. That reduces the side effects brought from other scheduled tasks to the lowest extent.

In a word, physmap-based attack is able to deal with much more kinds of use-after-free vulnerabilities in Linux kernel and have full control of data in freed memory. It has a wider application and higher stability. Two unavoidable disadvantages it suffers are memory cost and time cost, which can all be acceptable in practice.

5.2.4 Limitations

Not all the kernel objects which have use-after-free vulnerabilities are able to be overwritten by physmap-based attack on various platforms due to several reasons. Firstly, if a vulnerable object is going to be reused quickly after it has already been freed, then it is hard to re-occupy the object since the physmap spraying relatively takes time. And the immediate re-use of that object before overwriting is done may lead to kernel crashes. Secondly, if a vulnerable object is for internal use in kernel and it is not easy for an attacker to create a second instance of this kind of object, then even if we spray these vulnerable objects and place them uniformly in kernel space, the success rate of memory collisions based on physmap-based attack still decreases. Thirdly, when applying physmap-based attack, attackers require a certain amount of directly mapped memory to do spraying and make the physmap grow in order to touch vulnerable kernel objects. If the size of current usable physical memory is not enough, then the success rate of the attack goes down.

6. EVALUATION

6.1 Testing with Linux Kernel

In this section, we evaluate the performance of object-based attack and physmap-based attack by applying them to exploit our malicious kernel module mentioned in Section 2.1 in Linux kernel. The experiments are carried out on both 32-bit and 64-bit Ubuntu 14.04 with 2GB RAM. Note that when testing object-based attack based on collisions between objects of different sizes, the size of the vulnerable object in our module is set to be 576 as mentioned in Section 3.3.

Table 1 shows the success rate of object-based attack and physmap-based attack to compromise kernel by exploiting the malicious kernel module with basic memory requirements for different steps. Note that object-based attack of type 1 leverages memory collisions between objects of the same size and object-based attack of type 2 leverages memory collisions between objects of different sizes.

As shown in the experimental results in Table 1, object-based attack of type 1 performs better than object-based attack of type 2 as expected. Object-based attack of type 2 cannot be applied for a stable kernel exploitation due to its low success rate. And memory collision attack performs worse on 64-bit Linux platforms since much more entropy of the kernel memory layout is introduced. It can also be seen that physmap-based attack cost most memory compared to other types of attack in total. And it needs extra memory for spraying vulnerable objects and making them dispersedly allocated in kernel space.

As described before, object-based attack choose kernel objects as candidates to overwrite vulnerable objects. However, due to the limitations on resources a user cannot create too many kernel objects inside kernel. For example, a user can only create certain amount of socket connections, which means that limited *kmalloc-size* buffers during *sendmmsg* can be created in one time. Thus the memory requirement for kernel spraying in object-based attack has to in a certain

Attack Types	System	Memory Req. for Padding	Memory Req. for Bug-freed Objects	Memory Req. for Spraying	Success Rate
Object-based 1	32-bit	NaN	64KB	64KB	99%
	64-bit	NaN	96MB	128KB	80%
Object-based 2	32-bit	NaN	168KB	128KB	60%
	64-bit	NaN	160MB	256KB	40%
Physmap-based	32-bit	32MB	512KB	1536MB	99%
	64-bit	32MB	512KB	1536MB	85%

Table 1: Memory Collisions Attack in Linux kernel

range as shown in Table 1, which decreases stability of an attack.

Since the size of the vulnerable object in the malicious kernel module remains the same, both object-based attack and physmap-based attack take effect in the experiments. However, the strong powerful of physmap-based attack can be seen in the next section.

6.2 Testing with Android Kernel

In this section we evaluate our memory collision attack on Android devices with a use-after-free vulnerability (CVE-2015-3636) [4] which is credited to the author. We leverage this vulnerability to implement *PingPongRoot*, a universal exploit that achieves privilege escalation on most popular Android devices (Android version >= 4.3) including those with 64-bit processors.

6.2.1 PingPongRoot

PingPongRoot exploits the CVE-2015-3636 use-after-free vulnerability, which is related to a vulnerable PING sock object in the kernel. By specifying *sa_family* as *AP_UNSPEC* and making connections to a PING socket twice, the reference count of that PING sock object becomes zero, and thus the kernel frees it. However, that leads to a dangling file descriptor related to the PING sock object in a user program (The vulnerability can only be triggered on Android devices. For Linux PC, a common user does not have the privilege to create a PING socket). Therefore, attackers can operate on this file descriptor in a user program and make the kernel reuse the freed PING sock object, which leads to code execution in the kernel.

The vulnerable PING sock object has different sizes on different devices, thus *PingPongRoot* applies physmap-based attack instead of object-based attack to exploit such a vulnerability with high reliability. We demonstrate the exploitation of *PingPongRoot* on a representative Android device, Google Nexus 7 running Android Lollipop system. The exploit is conducted by following the steps described in Section 4. At first, D PING sockets are created for defragmentation. Then we iteratively spray PING sockets in group. For each group, every N PING sockets are allocated in newly-created processes. Note that all of these N PING sockets have to be created in other processes instead of the current process because of the resources limitation on each process. After these processes finish the allocating work, they hang up there not to cause kernel to release these PING sockets. Then M ($N \ll M$) PING sockets are created in the current process. These sockets are treated as vulnerable targets and later memory collisions will happen between these objects and the physmap.

According to physmap-based attack described in Section 4, all M PING sockets are released by triggering the use-after-free vulnerability. Then all the processes created at the beginning of our exploit are terminated. That causes kernel to recycle the resource of the terminated processes, thus all N PING sockets are freed by kernel allocators.

After that, *mmap* is repeatedly invoked. Each *mmap* allocates 256MB memory in user space. For every 8 dwords of the *mmaped* memory, the $7th$ dword is rewritten as a valid address in user space. All the other dwords are overwritten to be the magic values. By calling *ioctl* on those M vulnerable PING sockets with the argument *SIOCGSTAMPNS*, the member *sk_stamp* of a PING socket object can be read out to user space. Check it with pre-defined magic value to see whether a PING sock object is covered by the physmap or not. If no memory collision happens among those M ones with the physmap, back to the beginning of the exploit and spray more groups of padding objects with vulnerable objects. Otherwise, a vulnerable kernel object successfully covered by the physmap is achieved and the exploit continues.

In fact, the $7th$ dword of a PING socket object is the member *sk_prot*. It is a structure used to store the properties of PING protocol and it has a member which is a function pointer called *close*. When a PING socket is closed in a user program, then this function pointer is to be invoked. Due to the re-filling of the vulnerable PING socket object, the $7th$ dword of it is currently a virtual address in user space. That means the whole *sk_prot* structure is controlled by the attacker. The function pointers in the fake *sk_prot* are set to the address of kernel shellcode placed in user space. Then *close* is called on the dangling file descriptor related with that vulnerable object. The fake function pointer is invoked and our shellcode is executed in kernel context, which leads to a temporary privilege escalation on Android Lollipop.

6.2.2 Experimental Results

Our *PingPongRoot* exploit achieves privilege escalation on hundreds of Android devices and the performance of our attack is shown in Table 2.

The universal property and reliability of physmap-based attack is verified based on the following observations from Table 2. Firstly, most popular Android devices on market of all kinds of brands including **Samsung**, **SONY**, **Google Nexus**, **HTC**, **Huawei**, and **Xiaomi** are exploited by physmap-based attack with high root success rates. Note that the size of the vulnerable PING sock objects varies on different phones and tablets, but our generic exploit does not take that into considerations. Only RAM size affects our exploit settings. Secondly, the exploit settings are not changed much when attacking 64-bit Android kernel compared with the settings applied for attacking 32-bit Android kernel. Practically, physmap-based attack is verified to be effective also in 64-bit Android kernel. In a word, physmap-based attack is a universal and powerful solution to exploiting use-after-free vulnerabilities in Linux-based kernel.

System	RAM Size	Device	Memory for Padding	Memory for Spraying	Memory for Bug-freed Objects	Success Rate
Android 32-bit	1G	Huawei Honor 4 Xiaomi Hongmi Note	128MB	640MB	64KB	85%∼90%
	2G	Google Nexus 5/7 Samsung Galaxy S4/S5 HTC One M8 Huawei Mate 7/Ascend P7 Xiaomi M3	128MB	1024MB	64KB	98%
	3G	Samsung Galaxy Note 3 Sony Z2/Z3 Huawei Honor 6 Xiaomi M4	128MB	1536MB	64KB	98%
Android 64-bit	2G	Google Nexus 9	128MB	1024MB	64KB	90%
	3G	Samsung Galaxy S6/S6 Edge HTC One M9	128MB	1536MB	64KB	85%∼90%

Table 2: Performance of Physmap-based Attack on Android Devices

7. DEFENDING AGAINST MEMORY COLLISION ATTACK

One notable advantage of the physmap-based attack is that it firmly captures the inherent weakness of current versions of Linux kernel. Considering the direct mapping as a fundamental feature of the system, it is not easy to build highly effective mitigation against the physmap-based attack scheme. Two considerable approaches to defending against the attack model are presented as follows.

7.1 Bound for the Physmap

One effective way to defend against the memory collision attack through the physmap is to restrict the percentage of the total memory in use in the physmap for each user on the system.

For a particular user, all of his active tasks scheduled by Linux kernel are taken into consideration. And the sum of the memory in the user space of every task which is directly mapped by the physmap is recorded in the kernel. When this value arrives at a threshold value predefined by the kernel, then any more memory request for the physmap will be refused by the kernel unless the some of his task releases a certain amount of memory mapped by the physmap to make the total occupation of the physmap in the kernel lower than the threshold value.

Such an upper bound on the occupation of the physmap for all the active tasks owned by a user is able to defend against the memory collision attack through the physmap to a certain extent. In fact, in order to cover the SLAB caches storing the target kernel object, a large amount of memory has to be mmaped in user space by the active tasks of an attacker and sprayed with the data later used in the exploitation. However due to the threshold predefined by the kernel, such things cannot be achieved and it is not likely for the physmap to reach the relatively high virtual address where the target SLAB caches are allocated.

The restriction on the occupation of the physmap must be set for every user instead for every active task. That is because when attacking the core of the system, all the active tasks can influence the kernel memory. If the limitation on the occupation of the physmap is for every active task, attackers can create many independent processes and each of them follows the restriction but however the total amount of the memory mapped by the physmap in these processes is enough to cover the SLAB caches storing the target vulnerable objects and the protection is bypassed by the attacker. Thus, the attention should be put on the memory usage of all the active tasks owned by every user logging on the system.

As illustrated above, this protection can prevent attackers from making the memory collision attack through the physmap. And the effectiveness of this protection depends on that predefined threshold value. A lower threshold value providing more safety leads to the result that less directly mapped memory can be achieved by the user programs, thus a balance has to be found between the efficiency and the security of Linux kernel.

7.2 Complete Separation

The main cause of the memory collision attack through the physmap is that the directly mapped memory sourcing from the user space covers the place where originally the kernel objects are stored, a valid way to defend against the memory collision attack is to apply the complete separation between the physmap area and the SLAB caches.

The kernel specifies the memory of a fixed range to be the appropriative memory for the physmap. Even if a large part of the memory for special physmap use is free, it is impossible for the SLAB caches to be located in that region. Similarly, the physmap can never expand into the space once the kernel objects are stored there. A legible border is specified by this protection between the physmap area and the area for allocating objects in the kernel.

This protection against the collision attack is effective for sure since when it is on, there is no chance that a collision will happens between the physmap and the target vulnerable objects. However it is hard to be implemented on 32-bit Linux kernel because the size of the kernel space is too small to achieve a complete separation between the physmap and the SLAB caches. But for the 64-bit Linux kernel, the virtual address space is definitely enough for the kernel to achieve this. And without doubt, for safety, there will be more overhead and waste of the space.

8. RELATED WORK

In recent years, the use-after-free vulnerability become the most popular and serious vulnerability in the applications both on the desktop PC and on the mobile devices. In 2013, there were 129 CVEs for Microsoft Internet Explorer and most of them were use-after-free vulnerabilities, and this number raised to 243 in year 2014 [9]. Due to the various mitigations [34], exploiting other vulnerabilities like stack overflows and heap overflows has become harder. Taking advantage of the use-after-free vulnerability in the web browsers or in the document viewers, attackers are able to execute arbitrary code in the context of the applications and eventually control your computer [3] or mobile phone [2] remotely.

When exploiting a use-after-free vulnerability, the key step is to re-occupy the memory of the freed object [29] which means using some other objects to do the memory collision with the vulnerable object. Thus, attackers should carefully and accurately arrange the heap layout of the process [28], [32], [12] before triggering the use-after-free vulnerability.

In order to defend against the use-after-free vulnerability, numerous protection and mitigation schemes are proposed to enforce the temporal safety [29]. Many of these protections employ special allocators. It is the basic approach to protect against use-after-free exploits since it tries to prohibit the memory collision happens thus no re-use of the memory happens and the use-after-free vulnerability is hard to exploit. Such protections include Cling [13] and DieHarder [23]. In fact, almost all the current popular web browsers have their own allocators. They are known as *Heap Arena* [31] for WebKit in Safari on Mac OSX and iOS, *PartitionAlloc* [8] in Google Chrome, *Presentation Arena* and *Frame Poisoning* [21] in Mozilla Firefox and *Isolated Heap* [30] in Microsoft Internet Explorer. However, based on the rapid advancement of the exploit techniques, all of these allocators are defeated during the Pwn2Own/Pwnium hacking contests in the recent years due to their existed weakness. The related exploit techniques targeting the heap allocators of these web browsers include [15] for Safari, [5] for Google Chrome, [16] for Mozilla Firefox and [20] for Internet Explorer involving the well-designed heap layout arrangement, misalignment of the objects on the heap and utilizing the unique features of these private allocators.

Other protections against use-after-free include the object based approaches and the pointer based approaches [29]. The former ones are widely use in practice to detect memory corruptions include the *AddressSanitizer* [26] and the *Valgrind Memcheck* [11], trying to detect use-after-free by marking the memory once de-allocated. The latter ones focus on whether the pointer to be used is valid or not to detect use-after-free vulnerabilities. Such protections include [22, 19, 33].

By contrast, relatively fewer use-after-free vulnerabilities are discovered in OS kernel and only a few related researches are documented [24] [1]. Considering the requirement for the efficiency of the OS kernel, the protections like marking memory or checking dangling pointers cause an unacceptable overhead and cannot be adopted by the kernel. However, all the OS kernels have their special allocators. In Linux kernel there are SLAB/SLUB allocators which provide a perfect isolation between the objects of different sizes and different types. Thus it is difficult for an attacker to take advantage of a use-after-free vulnerability in Linux kernel and few exploit techniques attacking against the isolation of the kernel heap are documented.

Additionally, several previous reliable probabilistic techniques of exploiting the user application include [27] and [14]. These techniques also take advantage of the weakness of the memory management of Linux as well as our collision attack schemes do.

9. CONCLUSION

To reveal the universal exploiting solution for the use-after-free vulnerabilities in Linux kernel and show the security threat to the core brought by the memory recycling, in this paper a novel memory collision attack strategy is constructed and two memory collision attacks, the object-based attack and the physmap-based attack, are presented based on it. The attack strategy is widely applied and features the stability, complete bypassing the separation provided by the kernel allocator and the fully control of the re-filling data, and is proved to be effective on x86/x64, AArch32/AArch64 and Android Linux in practice. Particularly, a case study about rooting popular Android devices by exploiting CVE-2015-3636 with the physmap-based attack is detailed. Finally, two mitigation schemes are designed to to counter such memory collision attacks against Linux kernel with acceptable overhead.

10. ACKNOWLEDGEMENTS

We would like to thank our reviewers for valuable comments to improve the manuscript. We would also like to show our gratitude to Shi Wu, James Fang, Siji Feng, and Yubin Fu of Keen Team for their inspirations and great contributions to the development and related statistics of the universal Android root solution mentioned in this paper.

This work was supported in part by the National Key Technology Research and Development Program of China under Grants No. 2012BAH46B02, the National Science and Technology Major Projects of China under Grants No. 2012ZX03002011, and the Technology Project of Shanghai Science and Technology Commission under Grants No. 13511504000 and No. 15511103002.

References

[1] Attacking the Core: Kernel Exploiting Notes. `http://phrack.org/issues/64/6.html`.

[2] CVE-2010-1807. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1807`.

[3] CVE-2014-1776. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1776`.

[4] CVE-2015-3636. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636`.

[5] Exploiting 64-bit Linux like a boss. `http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html`.

[6] Exploiting NVMAP to escape the Chrome sandbox-CVE-2014-5332. `http://googleprojectzero.blogspot.com/2015/01/exploiting-nvmap-to-escape-chrome.html`.

[7] GCC stack protector support. `http://lxr.free-electrons.com/source/arch/x86/include/asm/stackprotector.h`.

[8] Google Chromium source. `https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h`.

[9] Microsoft Internet Explorer: CVE security vulnerabilities, versions and detailed reports.

[10] Short users guide for SLUB. `https://www.kernel.org/doc/Documentation/vm/slub.txt`.

[11] Understanding Valgrind memory leak reports. `http://es.gnu.org/~aleksander/valgrind/valgrind-memcheck.pdf`.

[12] J. Afek and A. Sharabani. Dangling Pointer: Smashing the Pointer for Fun and Profit. *Black Hat USA*, 2007.

[13] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proc. 19th USENIX Security Symposium*, 2010.

[14] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Proc. 35th IEEE Symposium on Security and Privacy*, 2014.

[15] L. Chen. WebKit Everywhere: Secure Or Not? *Black Hat Europe*, 2014.

[16] P. A. C. Karamitas. Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap. *Black Hat USA*, 2012.

[17] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proc. 23rd USENIX Security Symposium*, 2014.

[18] C. Lameter. Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB. *LinuxCon*, 2014.

[19] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proc. 2015 Annual Network and Distributed System Security Symposium*, 2015.

[20] J. Lu. New Exploit Mitigation In Internet Explorer. *HITCON*, 2014.

[21] MWR Lab. Isolated Heap & Friends - Object Allocation Hardening in Web Browsers. `https://labs.mwrinfosecurity.com/blog/2014/06/20/isolated-heap-friends---object-allocation-hardening-in-web-browsers/`.

[22] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. *ACM Sigplan Notices*, 2010.

[23] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *Proc. 17th ACM conference on Computer and communications security*, 2010.

[24] W. Robert. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proc. 1st USENIX Workshop on Offensive Technologies*, 2007.

[25] A. Rubini and J. Corbet. *Linux device drivers.* O'Reilly Media, Inc., 2001.

[26] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. 2012 USENIX Annual Technical Conference*, 2012.

[27] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM conference on Computer and communications security*, 2004.

[28] A. Sotirov. Heap feng shui in Javascript. *Black Hat Europe*, 2007.

[29] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal War in Memory. In *Proc. 34th IEEE Symposium on Security and Privacy*, 2013.

[30] TrendLabs. Isolated Heap for Internet Explorer Helps Mitigate UAF Exploits. `http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/`.

[31] G. Wicherski. Exploiting A Coalmine. *Hackito Ergo Sum*, 2012.

[32] T. Yan. The Art of Leaks: The Return of Heap Feng Shui. *CanSecWest*, 2014.

[33] Y. Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. 2015.

[34] Y. Younan, W. Joosen, and F. Piessens. Runtime Countermeasures for Code Injection Attacks against C and C++ programs. *ACM Computing Surveys*, 2012.