PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique

Yoochan Lee
Seoul National University
yoochan10@snu.ac.kr

Jinhan Kwak
Seoul National University
jinhan.kwak@snu.ac.kr

Junesoo Kang
UNIST
jsonbirth@unist.ac.kr

Yuseok Jeon *
UNIST
ysjeon@unist.ac.kr

Byoungyoung Lee *
Seoul National University
byoungyoung@snu.ac.kr

Abstract

The stealthiness of an attack is the most vital consideration for an attacker to reach their goals without being detected. Therefore, attackers put in a great deal of effort to increase the success rate of attacks in order not to expose information on the attacker and attack attempts resulting from failures. Exploitation of the kernel, which is a prime target for the attacker, usually takes advantage of heap-based vulnerabilities, and these exploits' success rates fortunately remain low (e.g., 56.1% on average) due to the operating principle of the default Linux kernel heap allocator, SLUB.

This paper presents PSPRAY, a timing side-channel attack-based exploitation technique that significantly increases the success probability of exploitation. According to our evaluation, with 10 real-world vulnerabilities, PSPRAY significantly improves the success rate of all those vulnerabilities (e.g., from 56.1% to 97.92% on average). To prevent this exploitation technique from being abused by the attacker, we further introduce a new defense mechanism to mitigate the threat of PSPRAY. After applying mitigation, the overall success rate of PSPRAY becomes similar to that from before using PSPRAY with negligible performance overhead (0.25%) and memory overhead (0.52%).

1 Introduction

The Linux kernel is a major attack target to accomplish the attacker's objectives (e.g., escalating privilege). To address these attempts, several kernel mitigation techniques (e.g., KASLR [11], KCFI [8, 12], and KDFI [23, 36]) have been proposed. These mitigation techniques make attacker's exploitation more difficult since the success rate of exploitation is a critical factor for attackers and developers.

For attackers, the stealthiness of an attack is one of the most important requirement when launching the exploitation. This is because if the exploitation fails, the attack would be caught by the defender. More specifically, such failure might lead to a kernel panic, which is a very apparent symptom that defenders can notice. Therefore, attackers are highly motivated to improve the success rate of their attacks. Even for kernel developers, if the vulnerability seems to be difficult to exploit, they may judge that the vulnerability is not a serious one. Therefore, developers often postpone the patching procedure so as to focus on other, more seemingly urgent ones.

Focusing on the popular heap-based kernel vulnerabilities (e.g., 968 of 1,107 vulnerabilities found by Syzkaller are heap-related vulnerabilities [40]), it is difficult for attackers to assure the successful exploitation. This is because, in order to exploit the heap-based vulnerabilities, the attacker should be able to roughly predict the current allocation status. However, it is nearly impossible for the following two reasons. First, the kernel's object allocation status is not accessible by attackers. As the privilege separation between the user and the kernel is employed, the user has no way to directly access the kernels' memory layout and thus to directly learn the allocation status. Second, the kernels' memory allocator shows non-deterministic, pseudo-random behavior, rendering it difficult to infer its internal status. To be specific, the allocation behavior of SLUB [7], the default Linux kernel's heap allocator, depends on various underlying contexts (e.g., an object order within the freelist is randomly determined, and the allocation pool is maintained with multi-stage pools in which each pool is dependent to other stage's pool).

To clearly showcase such difficulty, let us take examples using two representative heap vulnerabilities: out-of-bounds and use-after-free vulnerabilities. In the case of exploiting an out-of-bounds vulnerability, the attacker has to place two objects next to each other (i.e., one object triggering out-of-bounds and the other overwritten by the out-of-bounds). However, due to the slab freelist random [1] which is the default mitigation technique of major distributions (e.g., Ubuntu and Debian), the allocation order of the SLUB is pseudo-random, so the attacker cannot ensure such side-by-side object placements. Similarly, in order to exploit a use-after-free vulnerability, the attacker should be able to place two objects (i.e., one object being freed and the other being used/referenced) at the

*Corresponding authors

same virtual address. However, it is possible that the SLUB allocator may assign different allocation pools for these two objects so that there can be cases those two objects cannot be overlapped. We note that in order to increase reliability of the exploit, the heap spraying technique [9, 13, 35] has been popularly used. However, heap spraying alone cannot completely bypass the slab freelist random mitigation in the SLUB allocator.

This paper presents PSPRAY, a new kernel heap exploitation technique, which can significantly increase the reliability of the heap exploit against the slab freelist random mitigation technique. Acknowledging that the key to launching successful heap exploitation is learning the kernel allocation status, PSPRAY first develops new side-channel attacks against the SLUB allocator. Next, leveraging this new side-channel attack, we further show how to significantly augment the heap exploitation reliability.

More specifically, the key idea of PSPRAY is using the timing side-channel to indirectly learn the allocation status of the slab. To the best of our knowledge, while various timing side-channels have been presented [10, 15–17, 20, 24, 25, 34, 43], PSPRAY is the first timing side-channel attack for exploiting memory corruptions. Using the timing side-channel, we can differentiate the internal allocation path of SLUB, which in turn allows us to infer a partial allocation status (i.e., whether the allocation has taken place from the empty freelist or not).

Based on this allocation status information by PSPRAY, we further design new heap exploitation techniques. Specifically, when exploiting the out-of-bounds vulnerability, we find that attackers can avoid the cases where the vulnerable object and the target object are not adjacent. Likewise, in order to exploit use-after-free (as well as double-free), we find that attackers can learn that the vulnerable object and the target object are not placed at the same address, thereby avoiding the failure cases.

In order to demonstrate that PSPRAY is effective in augmenting the exploitation reliability, we developed exploits using PSPRAY for 10 real-world vulnerabilities. Our evaluation results showed that PSPRAY significantly improved the success rate, from 56.10% to 97.92% on average. Notably, in the case of 83bec2, which is found by Syzkaller [40], the exploitation with PSPRAY shows a 98.16% success rate while the exploitation without PSPRAY only showed a 13.70% success rate.

As a mitigation solution against PSPRAY's side-channel attacks, we further introduce a new defense mechanism. The basic idea is to obscure the timing when an empty freelist is used for allocation through irregular period assignments. Using this method, the success rate of attacks reverts back to what it was before using PSPRAY.

To summarize, this paper makes the following contributions:

- **Analysis of heap exploitation failure**. We figure out the cause of the exploitation failure through the operating principle of the SLUB allocator.
- **New heap exploitation technique.** PSPRAY presents a new Linux kernel heap exploitation method that bypasses the cause of the exploitation failure using timing side-channel attacks.
- **New mitigation technique.** We suggest and implement a new mitigation technique against PSPRAY.

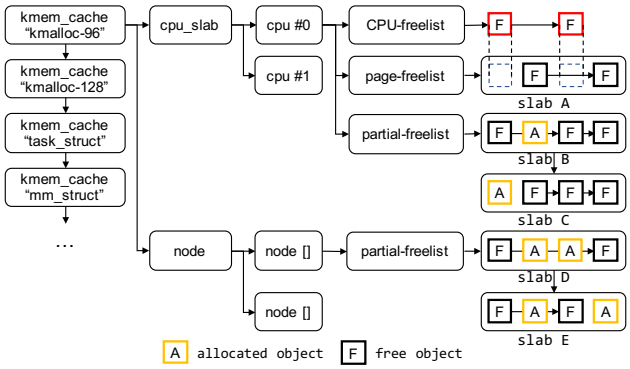


Figure 1: Architecture of SLUB allocator

2 Background

2.1 SLUB Allocator

SLUB allocator [7] is a default Linux kernel heap memory management mechanism intended for the efficient memory allocation of kernel objects. SLUB allocator manages the slabs, which are a group of one or more pages for fast and efficient object allocation.

The architecture of SLUB allocator. The basic architecture of the SLUB allocator is illustrated in Figure 1. The SLUB allocator includes various `kmem_cache` according to a specific type (e.g., `task_struct`) or a specific size (e.g., `kmalloc` series). Each `kmem_cache` uses both per-CPU mechanism and per-node mechanism for managing the slabs. More specifically, each CPU core has individual `freelist`, `page`, and `partial`. `Freelist` is a linked list of free objects on `page`, and every object allocated through SLUB allocate from this `freelist`. Even though there is a freelist for each slab, SLUB allocator manages a separate CPU-`freelist` in order to achieve faster performance. In more detail, accessing the CPU's `page`-freelist or CPU's `partial`-freelist after accessing target slab is slower than immediately accessing the CPU-`freelist`. The CPU's `page` consists of one slab and `partial` consists of one or more slabs. SLUB allocator manages free objects with linked lists. As previously stated, each slab has an individual freelist separate from the CPU-`freelist`, and these freelists are used when the CPU-`freelist` is empty. Also, each node has a `partial`, which consists of one or more slabs. Each node's `partial`-freelist can also be used when CPU-`freelist` is empty.

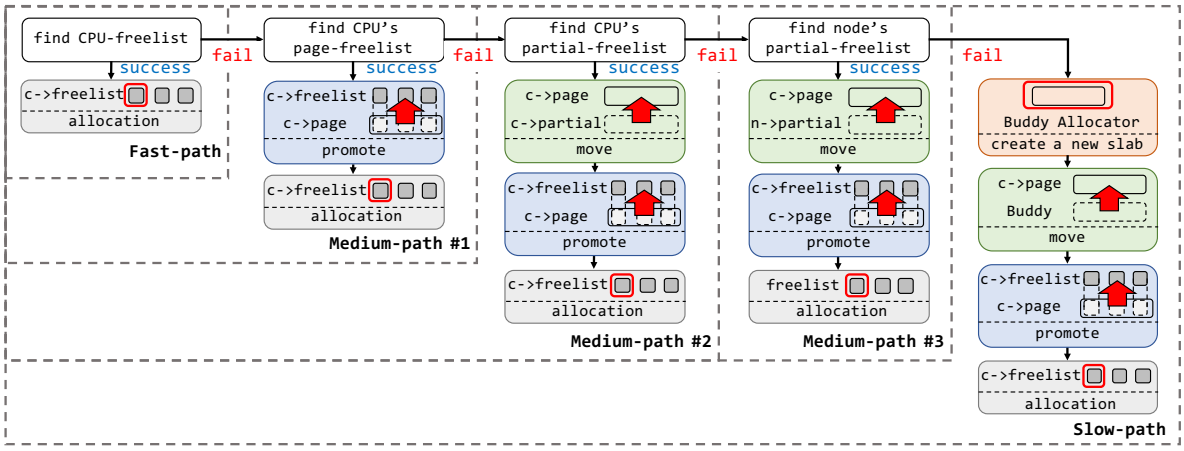


Figure 2: Allocation sequence of SLUB allocator

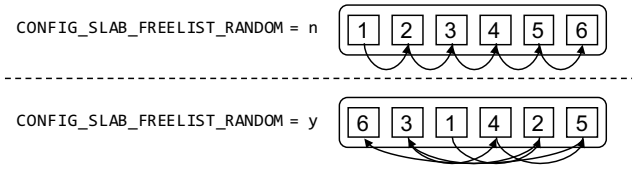


Figure 3: Mechanism of slab freelist random

Allocation sequence of SLUB allocator. Figure 2 shows the allocation sequence of SLUB allocator. A rule of thumb is that all objects are allocated from CPU-`freelist`. If CPU-`freelist` is not available, then CPU-`freelist` must become available first by reclaiming from other slabs. Therefore, the object allocation can take different paths depending on whether the object allocation is immediately performed on CPU-`freelist` or performed after reclaiming.

The fast-path is the case where there is an available object in the CPU-`freelist`, and thus it allocates the object immediately. If there is no object in the CPU-`freelist`, the medium path #1 is taken, in which the kernel promotes the CPU's `page`-freelist to the CPU-`freelist` and then allocates the object. If the CPU-`freelist` and the CPU's `page` are empty, the SLUB allocator executes the medium path #2, in which it moves from CPU's `partial` to CPU's `page` and promotes the CPU's `page`-freelist to CPU-`freelist`. Even if there is no slab in both CPU's `page` and CPU's `partial`, the SLUB allocator executes the medium path #3, in which it moves from node's `partial` to CPU's `page` and promotes the CPU's `page`-freelist to CPU-`freelist`. Lastly, the slow-path is the case where there is no slab in CPU's `page`, `partial`, and node's `partial`. Therefore, the SLUB allocator creates a new slab using buddy allocator. Then, the kernel assigns a new slab to CPU's `page` and promotes the CPU's `page`-freelist to the CPU-`freelist`.

2.2 Slab Freelist Random

The Linux kernel offers a configuration option, `CONFIG_SLAB_FREELIST_RANDOM` [1], designed to mitigate the OOB vulnerability exploitation. In particular, it is a Linux security configuration which has been added since Linux `v4.8`, and major distributions such as Ubuntu and Debian have been using it by default since Ubuntu `v16.04` and Debian `v9`, respectively.

Figure 3 illustrates before and after this configuration is applied. Before this configuration is enabled, the SLUB allocator allocates the object from the slab sequentially. Therefore, one can easily place the vulnerable object and the target object, which contains either pointer or critical data, next to each other when exploiting OOB vulnerability. After this configuration is applied, however, the SLUB allocator randomizes the order of the freelist when the kernel creates the new slab. As a result, it becomes difficult for the attacker to place vulnerable object and target object next to each other, thereby hardening the kernel from the OOB exploitation.

3 Exploitation method and failure cases

In general, the memory corruption arising from the kernel heap area can be divided into three vulnerability types: out-of-bounds, use-after-free, and double-free. In this section, we first explain how to exploit each vulnerability type. Then, we further show how the exploitation attempts can fail due to the random nature of SLUB allocator.

3.1 Out-Of-Bounds

Exploitation method. Out-Of-Bounds (OOB) is a vulnerability accessing beyond a predetermined heap object size (e.g., a heap buffer overflow). In order to exploit the OOB, an attacker should place two objects, a vulnerable object and a target object, next to each other. The vulnerable object is an object that can trigger out-of-bounds access. The target object is an object which is overwritten by the OOB. It should contain a function pointer or other critical data, such that once overwritten, it would allow the attacker to divert control- or data-flows.

When exploiting the OOB in the random nature of SLUB allocator, the most common technique to place the vulnera-

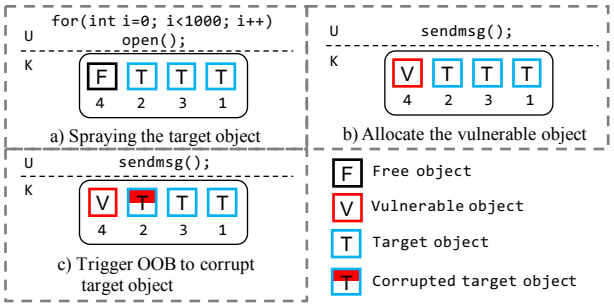


Figure 4: The exploitation process of CVE-2017-7533 (OOB)

ble object and target object next to each other is heap spraying [9, 35], which is also known as heap grooming [13]. Heap spraying is the technique that transforms the heap from an unknown and random state into a partially known state. However, the attacker cannot know the information about freelist order so that the heap spraying can allocate from zero to N-1 objects in one slab with N being the number of objects per slab. Then, the attacker allocates the vulnerable object. Since the allocation of the target object is uniformly random, if the attacker is lucky, the vulnerable object and the target object are allocated adjacent to each other.

Figure 4 shows the success case when exploiting CVE-2017-7533. The attacker first allocates the target object 1,000 times using the syscall, `open()` (in Figure 4-a). Next, the attacker allocates the vulnerable object using `sendmsg()` syscall (in Figure 4-b). If the attacker is lucky, the vulnerable object and the target object are adjacent to each other. Finally, if the attacker triggers the OOB vulnerability using another syscall, `sendmsg()`, then the target object is corrupted (in Figure 4-c).

Failure cases. Due to the random nature of SLUB allocator, the exploitation attempt may fail when exploiting the OOB (shown in Figure 5). The reason is that it is difficult for the attacker to place the vulnerable object and the target object next to each other because of slab freelist random and the fact that the attacker (granted with the user's privilege) cannot know the order of freelist. Therefore, the attacker cannot know how many target objects have been filled up in the slab when using heap spraying. If the number of target objects allocated to the slab is zero, it fails (in Figure 5-a). If the number of target objects allocated to the slab is one or two, there are cases where it fails depending on the freelist order (in Figure 5-b, c). If the number of target objects allocated to the slab is three, the vulnerable object is allocated to the last position of the slab, and it fails (in Figure 5-d).

Probability model. The probability of successful OOB exploitation is as follows. We assume that the number of objects that can be allocated in one slab is N, and the order of freelist is uniformly random. This assumption is reasonable because the Linux kernel uses the Fisher-Yates shuffle algorithm [14] to randomize the slab freelist. We further assume that one vulnerable object and k sprayed target objects are allocated in the same slab at random. To be a successful exploitation, a sprayed target object must be allocated right after the vul-

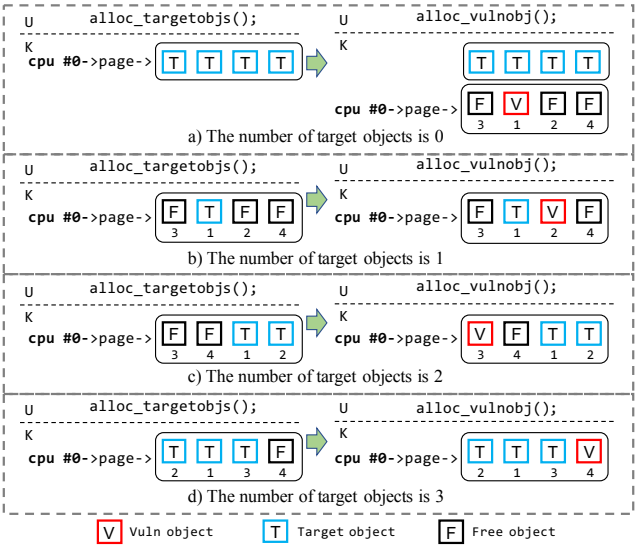


Figure 5: OOB vulnerability exploitation failure case

nerable object. The probability that the vulnerable object and sprayed target object are adjacent in the same page is represented by the following formula:

$$\frac{{}_{N-1}C_k * {}_kC_1}{{}_NC_k * {}_{N-k}C_1} = \frac{k}{N}$$

For the entire case, we choose k target objects from N free objects and then choose one vulnerable object from remaining free objects. For the exploitable case, the vulnerable object and target object must be adjacent. Therefore, we choose k target objects from $N-1$ free objects (except one object from N free objects for vulnerable object). Then, we choose one vulnerable object from selected k target objects. Furthermore, we calculate the average probability of each k case. The number of sprayed target objects can be from 0 to $N-1$, so the probability of OOB exploitation with a random slab freelist is calculated with the following formula:

$$P_{\text{OOB}}^{Baseline} = \frac{\sum_{k=0}^{N-1} \frac{k}{N}}{N} = \frac{N-1}{2N}$$

3.2 Use-After-Free and Double-Free

UAF exploitation method. Use-After-Free (UAF) is a vulnerability if the object is accessed after being freed. To exploit UAF, an attacker needs to place the vulnerable object (i.e., being freed) and target object (i.e., being allocated after freeing) at the same virtual address. Figure 6 shows the exploitation process when exploiting the CVE-2019-2215. The attacker first executes the syscall, `epoll_ctl()`, which allocates both the vulnerable object and one additional object in kmalloc-512 (in Figure 6-a). Then, the attacker executes the syscall, `ioctl()`, which frees the vulnerable object (in Figure 6-b).

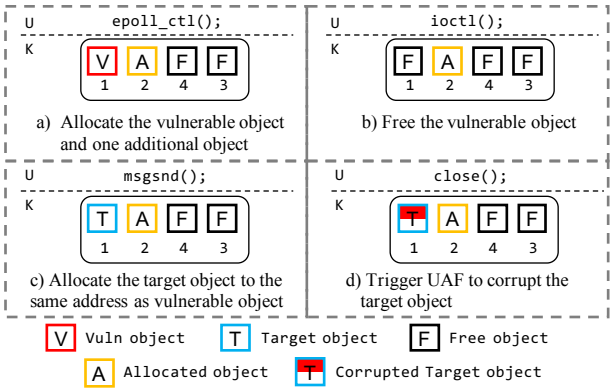


Figure 6: The exploitation process of CVE-2019-2215 (UAF)

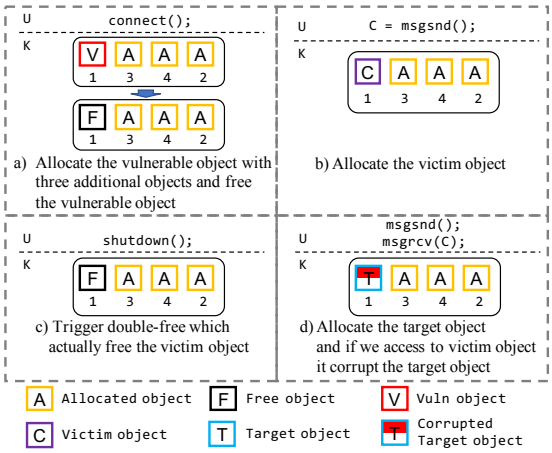


Figure 7: The exploitation process of CVE-2017-6074 (DF)

Next, executing the syscall, `msgsnd()`, allocates the target object to the same virtual address, which was originally placed by the vulnerable object (in Figure 6-c). Finally, the attacker execute the syscall, `close()`, which accesses the freed vulnerable object (in Figure 6-d).

DF exploitation method. The Double-Free (DF) vulnerability frees an object that has already been freed. In order to exploit the DF vulnerability, two objects (i.e., victim object and target object) are needed. The attacker allocates and frees the vulnerable object using first free. Then, the attacker allocates the victim object at the same address as vulnerable object and frees the victim object using double-free. At this time, because the attacker frees the victim object with the code of freeing the vulnerable object, it leaves the pointer (i.e., dangling pointer) that points to the victim object. Then, the attacker allocates the target object at the same address as vulnerable object and victim object. Finally, if the attacker accesses the victim object using dangling pointer, it corrupts the target object. Figure 7 shows the process when exploiting the CVE-2017-6074. The attacker first allocates the vulnerable object and three additional objects in kmalloc-2048 and frees the vulnerable object using the `connect()` syscall (in Figure 7-a). Then, the attacker allocates the `msg_msgseg` object as the victim object using the `msgsnd()` syscall (in Figure 7-b). Then, the `shutdown()` system call is executed, which frees

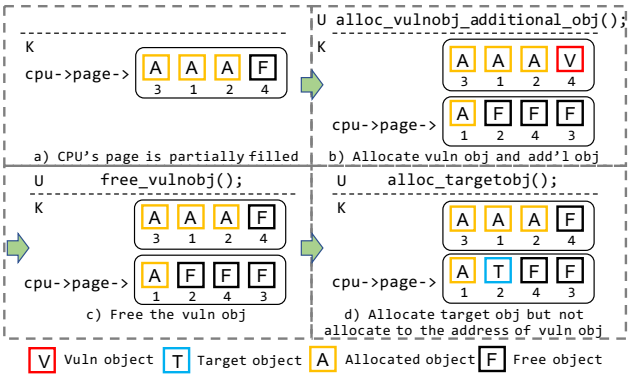


Figure 8: UAF and DF vulnerability exploitation failure case

the already freed vulnerable object (in Figure 7-c). In reality, it frees the victim object and leaves dangling pointer, which points to the victim object. Then, the attacker allocates the `msg_msg` object to the same address as the target object using `msgsnd()` system call (in Figure 7-d). At this time, the attacker can access and corrupt the target object using a dangling pointer.

Failure cases. Due to the inherent operational behaviors of the SLUB allocator, there are restrictions when exploiting the UAF and DF vulnerabilities. Figure 8 shows one reason (changed CPU's `page`) for failure when exploiting UAF and DF vulnerabilities.

Depending on the vulnerability, a system call that allocates a vulnerable object does not always allocate only one object. In the case of CVE-2018-6555, it allocates the vulnerable object with 12 additional objects in kmalloc-96. The attacker supposes that the CPU's `page` is partially filled (in Figure 8-a). Then, the attacker executes the system call that allocates vulnerable object and additional object. When allocating the vulnerable object, the slab, which is assigned to CPU's `page`, is fully filled so that the corresponding slab is moved to full-list and the CPU's `page` is emptied. Then, when allocating the additional object, the kernel executes medium-path #2, #3 or slow-path to fill the CPU's `page`. That is, CPU's `page` is changed to another slab, and the additional object is allocated from the changed CPU's `page` (in Figure 8-b). Then, the attacker frees a vulnerable object (in Figure 8-c). Lastly, if the attacker tries to allocate the target object, it cannot reallocate to the address of the vulnerable object. Because the CPU's `page` is changed, the kernel allocates the target object from the changed CPU's `page` (in Figure 8-d).

Probability model. The success rate of UAF and DF exploitation is as follows: N denotes the number of objects per slab and A denotes the number of object allocations by the system call, which allocates the vulnerable object and additional objects.

$$P^{Baseline}_{\text{UAF \& DF}} = \begin{cases} \frac{N-A+1}{N} & (\text{if } A < N) \\ \\ \frac{1}{N} & (\text{if } A \geq N) \end{cases}$$

It can be divided into two cases. If A is less than N, the suc-

cess rate will depend on whether the vulnerable object and additional objects are placed in one slab. Otherwise, if A is greater than or equal to N, the exploitation will be successful when the last object allocated by the system call, which allocates the vulnerable object, fully fills the CPU's `page`. If the slab is fully filled up, the corresponding slab is moved to the full-list that manages the fully allocated slab, and then the CPU's `page` is emptied. At this time, if the object is freed, the slab that contains this object is assigned to the page of the CPU again. In other words, it is possible to reallocate the target object to the same address as the vulnerable object.

4 Our Approach : PSPRAY

The major reason for exploitation failure is that the information about slab's allocation status is not available to the attacker. Thus, the random nature of the SLUB allocator provides effective statistical mitigation against the attacker. However, we find the timing side-channel, which allows us to learn the allocation status of the slab partially. Specifically, using this timing side-channel, the attacker can notice the new and clean-state slab is created, which in turn allows inferring the allocation status of the corresponding slab. Therefore, the attacker can avoid exploitation failure cases based on this additional information.

4.1 Timing Side-Channel on SLUB allocator

As we have seen in §2.1, the SLUB allocator allocates the object with five different paths (i.e., fast-path, medium-path #1, #2, #3, and slow-path). The reason why the SLUB allocator is divided into these multiple paths is that it is optimized to improve performance. The SLUB allocator uses a fall-through algorithm to execute an efficient path with as little overhead as possible (as shown in Figure 2).

To clearly show the different performance of each path, we measure the performance from the beginning of `slab_alloc_node()`, which is the main function of `kmalloc()`, to the end of `slab_alloc_node()` using `msgsnd()` system call, which allocates one object (more details in §4.3). Then, we take the average of 100 runs for each path, which shows noticeable differences per path. The fast-path, which allocates the object from the `freelist` of CPU, is the fastest (on average 459 cycles). The medium path #1, #2, and #3 show 676, 1,191, and 1,848 cycles, respectively. On the contrary, the slow-path is the slowest, which shows a significant performance gap with the other paths (on average 6,048 cycles). This is because the slow-path creates a new slab using buddy allocator, moves a new slab to CPU's `page`, and promotes CPU's `page`-freelist to CPU-`freelist`, which takes a considerable amount of time. This implies that, if the attacker can precisely measure the performance of allocation, an attacker can tell which path has been taken for the allocation, establishing the timing side-channel on SLUB allocator's internal behaviors. Then, the question becomes based on the knowledge of which path has been taken for the allocation. How can one further infer the allocation status?

4.2 Inferring Allocation Status

Even if one can differentiate each path, not all paths are useful. To avoid exploitation failure, we need to learn the allocation status of the slab (e.g., how many objects are allocated in the target slab). However, fast-path and medium path #1 use the CPU's `page`, so we cannot tell the allocation status of the slab. Moreover, the medium path #2 and #3 move the CPU's and node's `partial` to CPU's `page`. However, the `partial` usually has one or more already allocated objects. As such, it is difficult to infer the slab's allocation status if those paths are taken.

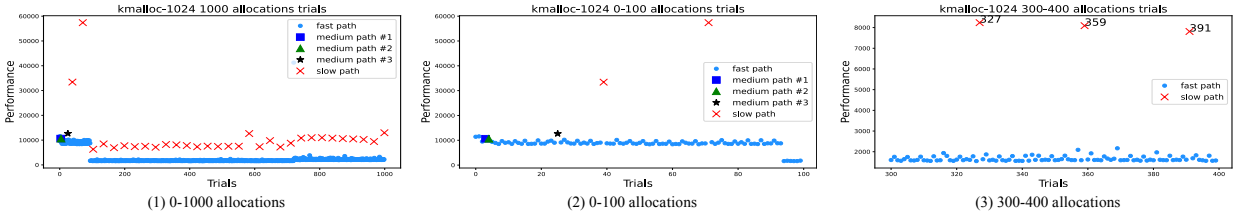
However, the behavior of the slow-path is unique from other paths. If the slow-path is taken, the kernel creates a new slab from the buddy allocator and allocates one object from the new slab. Therefore, we can tell whether the currently used slab is a new slab and the allocation status of the corresponding slab is filled only with one object. In other words, if we distinguish between the slow-path and other paths through the timing side-channel, we can predict the allocation status of a new slab.

4.3 Proof-Of-Concept

Finding an adequate system call. In order to use timing side-channel, we need a system call that is used for object allocation and has three conditions: First, the system call should be available as a user privilege. Second, the system call must allocate one object. If the system call allocates more than one object, it would be difficult to infer which path has been taken. Third, the system call should not have a significant impact on performance except for object allocation.

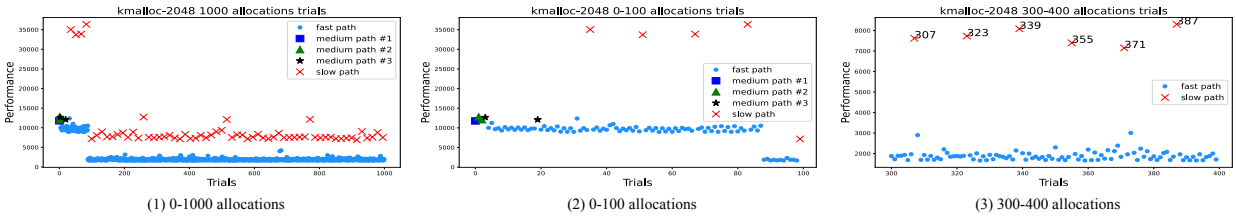
To find an adequate system call, we modify the Linux kernel to trigger panic when the system call allocates one object in kmalloc-series. Then, we do the fuzz testing using Syzkaller [40] for 24 hours and find 23 system calls that meet these three conditions (as shown in Table A.1). These system calls allocate object and copying the data from user space, while the rest code does not critically affect performance. Using these system calls, we can cover all of the kmalloc series (i.e., from kmalloc-32 to kmalloc-8192).

Experiment. To prove that the effectiveness of the theoretical timing side-channel on SLUB allocator is possible in the real-world, we allocate the kernel object 1,000 times using `msgsnd()` systems call, which is prevalently used for Linux kernel heap spray. At the same time, we measure the performance of `msgsnd()` system call from user space. The kernel object allocation is performed through a total of three experiments for each of the three `kmem_cache` (i.e., kmalloc-1024,



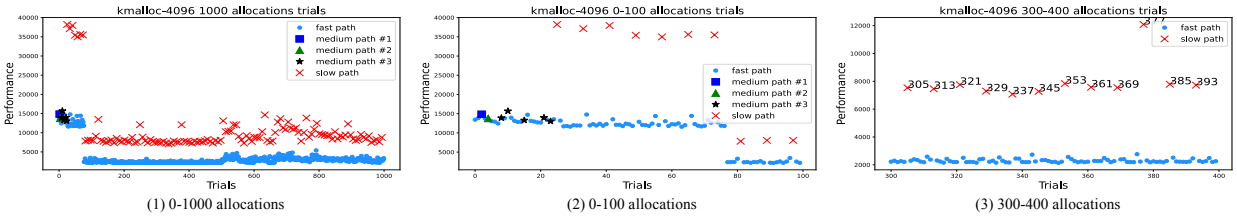
(1) 0-1000 allocations (2) 0-100 allocations (3) 300-400 allocations

(a) The value of measuring performance in `kmalloc-1024`



(1) 0-1000 allocations (2) 0-100 allocations (3) 300-400 allocations

(b) The value of measuring performance in `kmalloc-2048`



(1) 0-1000 allocations (2) 0-100 allocations (3) 300-400 allocations

(c) The value of measuring performance in `kmalloc-4096`

Figure 9: The value of measuring performance using `msgsnd()` system call

kmalloc-2048, and kmalloc-4096). Figure A.1 is the source code used in the experiment. We experiment on a machine with Ubuntu 20.04.3 LTS on Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 32G RAM, and 512GB HDD.

Experiment Results. Figure 9 shows performance evaluation result using `msgsnd()` system call from user space. Figure 9(a)-1, Figure 9(b)-1, and Figure 9(c)-1 show the performances of the total 1,000 allocations in kmalloc-1024, kmalloc-2048, and kmalloc-4096. In the first 100 allocations, as illustrated in Figure 9(a)-2, Figure 9(b)-2, and Figure 9(c)-2), the fast-path and medium paths are hard to differentiate. This is because `msgsnd()` not only allocates objects through `kmalloc()` but also copies data through `copy_from_user()`.

However, slow-path shows a noticeable performance difference from other paths. Before the code of `msgsnd()` is loaded to the CPU cache, the average performance of other paths is about 10,000 cycles, and the average performance of slow-path is about 35,000 cycles. After around 80 trials, the code of `msgsnd()` is loaded to the CPU cache, which is more faster and stabler. Therefore, looking at trials from 300 to 400 (in Figure 9(a)-3, Figure 9(b)-3, and Figure 9(c)-3), it can be seen that fast-path and slow-path can be intuitively distinguished. In conclusion, in the user space, we can differentiate the fast-path and slow-path through the timing side-channel so that we can notice whether a new slab is created and the allocation status of the corresponding slab.

5 Application of PSPRAY

This section introduces applications of PSPRAY to avoid the exploitation failure due to the random nature of the SLUB allocator.

5.1 OOB Exploitation

As previously stated in §3.1, the main cause of failure when exploiting OOB vulnerability is that slab freelist is random. SLUB allocator randomizes the freelist order when the slab is created. It lowers the probability the adjacent allocation of the vulnerable object and the target object. However, using PSPRAY, we can predict the allocation status of the slab. By applying this clue, we can fill the slab with one vulnerable object and then fill the remaining area with a target object.

Figure 10 shows the application method of PSPRAY, which bypasses the randomness of slab freelist. At first, using PSPRAY, we can find when the slow-path is executed (in Figure 10-b). Then, the slab B is created, and one object is allocated to the slab B. If the number of objects per slab is N, then we allocate the sprayed object $N-1$ times to fill up the slab B (in Figure 10-c). Now, if we allocate an object, then the kernel executes the slow-path, which creates a new slab C for object allocation. Therefore, we allocate the target object $N-1$ times to fill everything but one object in the new slab

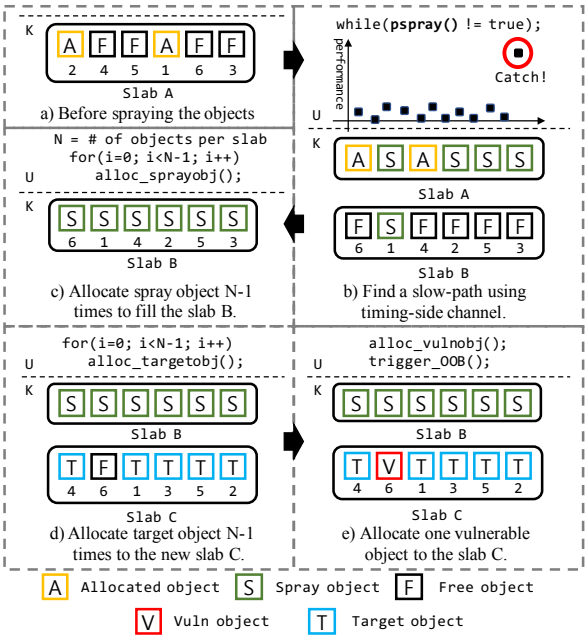


Figure 10: Exploiting OOB with PSPRAY.

(in Figure 10-d). Then, we allocate one vulnerable object to fill up the new slab C (in Figure 10-e). At this point, the slab C will be filled with $N-1$ target objects and one vulnerable object. If the vulnerable object is not in the last position of the slab, the OOB vulnerability will corrupt one of the target objects. Therefore, the success rate of this exploitation without noise (e.g., if the other process allocates the object to the same slab right before allocating the vulnerable object) is as follows:

$$P_{\text{OOB}}^{\text{PSPRAY}} = \frac{N-1}{N}.$$

5.2 UAF and DF Exploitation

As previously mentioned in §3.2, the main cause of failure when exploiting the UAF and DF vulnerability is that the CPU's `page` is changed to another slab. However, using PSPRAY, we can circumvent this situation.

Figure 11 shows the method of exploiting UAF and DF vulnerabilities using PSPRAY. We first use PSPRAY to find when a new slab is created through slow-path (in Figure 11-b). Therefore, we know the current CPU's `page` is filled with one object. Next, we allocate the vulnerable object and additional objects (in Figure 11-c), and then we free the vulnerable object (in Figure 11-d). After that, we allocate the target object to the same address as the vulnerable object (in Figure 11-e). Lastly, if we trigger the UAF or DF, it will corrupt the target object. Therefore, the success rate of exploitation using PSPRAY, without noise, is as follows.

$$P_{\text{UAF \& DF}}^{\text{PSPRAY}} = 100\%.$$

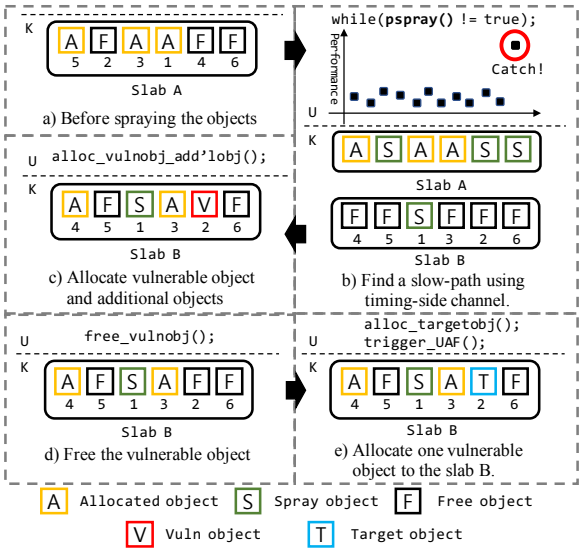


Figure 11: Exploiting UAF with PSPRAY.

6 Attack Evaluation

This section aims at evaluating the exploitation effectiveness of PSPRAY. To verify the effectiveness of PSPRAY, we divide the evaluation into two steps. First, we develop and exploit synthetic Out-Of-Bounds, Use-After-Free, and Double-Free vulnerabilities (§6.1) to show that PSPRAY is effective in most `kmem_cache` (i.e., from kmalloc-64 to kmalloc-4096). Second, we exploit 10 real-world vulnerabilities with PSPRAY (§6.2) to verify real-world impact.

Environment Setting. All of our experiments are performed on a server running Ubuntu 18.04.5 LTS with Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz (48 cores in total) and 512GB RAM. For each exploit, we compile the Linux `v4.15` and create disk image. Then, we run it in `QEMU 2.11.1` [4] virtual machine (VM) with 2 CPU cores and 2GB RAM VM configuration.

6.1 Synthetic Vulnerability

6.1.1 Design Synthetic Vulnerability

To conduct an in-depth study on the effectiveness of PSPRAY, we create synthetic vulnerabilities of OOB, UAF, and DF for Linux. These vulnerabilities are added to a plain Linux `v4.15` kernel source code. For these OOB, UAF, and DF vulnerabilities, we use three different system calls to allocate the vulnerable objects and trigger the vulnerabilities. We carefully implement an OOB read vulnerability and a UAF read vulnerability to prevent the kernel panic during the experiment because if the kernel panic occurs, the system is stopped, and the result of the rest of the trials is affected. For UAF and DF vulnerability exploitations, we collect and analyze six real-world vulnerabilities. Based on our analysis, we realize that these system calls allocate a vulnerable object with four

Cache	# of obj per slab	OOB				UAF with 4 dummy objects				DF with 4 dummy objects			
		$P_{OOB}^{Baseline}$	Baseline	P_{OOB}^{PSPRAY}	PSPRAY	$P_{UAF}^{Baseline}$	Baseline	P_{UAF}^{PSPRAY}	PSPRAY	$P_{DF}^{Baseline}$	Baseline	P_{DF}^{PSPRAY}	PSPRAY
kmalloc-64	64	49.21%	41.48%	98.43%	96.81%	94.00%	93.27%	100.00%	100.00%	94.00%	95.13%	100.00%	100.00%
kmalloc-96	42	48.80%	42.27%	97.62%	97.83%	90.47%	91.25%	100.00%	100.00%	90.47%	91.53%	100.00%	100.00%
kmalloc-128	32	48.43%	43.36%	96.87%	96.86%	87.50%	86.89%	100.00%	100.00%	87.50%	87.49%	100.00%	100.00%
kmalloc-192	42	48.80%	33.61%	97.61%	95.20%	90.47%	89.98%	100.00%	100.00%	90.47%	90.31%	100.00%	100.00%
kmalloc-256	32	48.43%	17.94%	96.87%	95.04%	87.50%	88.13%	100.00%	100.00%	87.50%	87.83%	100.00%	100.00%
kmalloc-512	32	48.43%	37.34%	96.87%	95.54%	87.50%	87.32%	100.00%	100.00%	87.50%	88.12%	100.00%	100.00%
kmalloc-1024	32	48.43%	27.28%	96.87%	96.44%	87.50%	88.13%	100.00%	100.00%	87.50%	86.98%	100.00%	100.00%
kmalloc-2048	16	46.87%	45.41%	93.75%	92.90%	75.00%	74.75%	100.00%	100.00%	75.00%	75.13%	100.00%	100.00%
kmalloc-4096	8	43.75%	26.70%	87.50%	87.04%	50.00%	51.46%	100.00%	100.00%	50.00%	50.59%	100.00%	100.00%

Table 1: Exploitation results on synthetic vulnerabilities

additional objects on average. Therefore, we evaluate the UAF and DF vulnerabilities assuming that a system call usually allocates a vulnerable object with four additional dummy objects. In addition, each experiment consists of 20 rounds and each round attempts the exploitation 500 times.

OOB exploitation. Figure A.2 shows the code of synthetic OOB read vulnerability. This code is implemented similarly to CVE-2016-6187. When a system call is called, an object is first allocated through `kzalloc()`. Then, the kernel triggers an OOB read vulnerability that reads the next object's data. If the value read through the OOB read is the nonce value that we sprayed through `msgsnd()`, the attack will succeed.

Experiments on this vulnerability are conducted in two methods. The first method is to use the heap spraying, which is introduced in §3.1. Then, the second method is to use PSPRAY, which is described in §5.1.

UAF exploitation. The code of a synthetic UAF read vulnerability, which is implemented similarly to CVE-2018-6555, appears in Figure A.3. In this example code, this system call (i.e., `uaf_test()`) provides three commands. If cmd is 1, the kernel allocates the vulnerable object, which is the vulnerable object used for UAF read, along with four additional dummy objects. If cmd is 2, the kernel frees the vulnerable object. Then, if cmd is 3, the kernel reads the value from the vulnerable object, which is freed at cmd 2.

DF exploitation. In Figure A.4, one can see the code of synthetic Double-Free vulnerability. This code is implemented similarly to CVE-2017-6074. In this code, this system call provides two commands. If cmd is 1, the kernel allocates the vulnerable object and allocates four dummy objects. If cmd is 2, the kernel frees the vulnerable object.

Experiments on these UAF and DF vulnerabilities are also conducted in two methods: The first method is the general approach when exploiting UAF and DF vulnerabilities, which are described in §3.2. The second method is using PSPRAY which, is described in §5.2.

6.1.2 Synthetic Vulnerability Exploitation Results

As shown in Table 1, PSPRAY shows a higher success rate than regular exploitation in all cases. In the case of the OOB vulnerability, our probability model (i.e., $P_{OOB}^{Baseline}$) shows an average of 48%, but the real success rate that we measured using heap Feng Shui shows at least 17.94%, which is lower than our probability model. This is because the noise can occur during the allocation of the target object or right before the allocation of the vulnerable object. This leads to hindering the vulnerable object and the target object from being adjacent to each other. On the other hand, exploitation using PSPRAY shows an average of 94.61%, which is similar to the expected success rate (i.e., P_{OOB}^{PSPRAY}). The reason why PSPRAY does not show a nearly perfect success rate is the case where the vulnerable object is allocated to the last location of the slab, which cannot corrupt the target object even if the OOB is triggered.

As for the UAF vulnerability case, the measured success rate shows 83.46% on average, and each case is quite similar to the probability model (i.e., $P_{UAF}^{Baseline}$). This is because there is only one case in which noise occurs and interferes with the exploit (i.e., right before the allocation of the target object). More specifically, the lowest success rate is kmalloc-4096, showing a 51.46% success rate. This is because the number of objects per kmalloc-4096 slab is eight. Therefore, if more than four objects are already allocated in the slab, the allocation of the vulnerable object and four additional dummy objects make the slab fill up, causing the CPU's `page` to change. However, the exploitation using PSPRAY has a 100% success rate in all cases, which is the same as our probability model (i.e., P_{UAF}^{PSPRAY}). This is because using PSPRAY, we can avoid CPU's `page` is changed.

In the case of the DF vulnerability, the measured success rate shows 83.67%, which is similar to our probability model (i.e., $P_{DF}^{Baseline}$). Similar to UAF, there are two cases where noise can be generated (i.e., right before the allocation of the victim object and target object). The lowest success rate is kmalloc-4096, which shows a 50.59% success rate. This is because if more than four objects are already allocated in the slab, the allocation of the vulnerable object and four additional dummy objects makes the slab fill up, which changes the CPU's `page`. That is, it fails. However, using PSPRAY, the success rate of exploitation shows 100% in all cases. This is the same as our probability model (i.e., P_{DF}^{PSPRAY}).

CVE	Bug Type	Slab Cache	# of obj per slab	# of alloc	$P_{\text{OOB}}^{Baseline}$	Baseline (Idle)	Baseline (Busy)	$P_{\text{OOB}}^{\text{PSPRAY}}$	PSPRAY (Idle)	PSPRAY (Busy)
CVE-2017-7533 [31]	OOB	kmalloc-256	32	2	48.43%	33.78%	25.64%	96.87%	94.26%	90.48%
CVE-2017-7184 [30]	OOB	kmalloc-128	32	1	48.43%	21.18%	18.86%	96.87%	96.52%	91.86%
CVE-2016-6187 [28]	OOB	kmalloc-128	32	1	48.43%	23.38%	20.18%	96.87%	95.58%	92.12%
CVE-2010-2959 [27]	OOB	kmalloc-512	32	2	48.43%	39.60%	24.08%	96.87%	94.80%	92.60%
CVE	Bug Type	Slab Cache	# of obj per slab	# of alloc	$P_{\text{UAF \& DF}}^{Baseline}$	Baseline (Idle)	Baseline (Busy)	$P_{\text{UAF \& DF}}^{\text{PSPRAY}}$	PSPRAY (Idle)	PSPRAY (Busy)
CVE-2019-2215 [33]	UAF	kmalloc-512	32	2	96.87%	93.28%	91.18%	100.00%	100.00%	99.98%
CVE-2018-6555 [32]	UAF	kmalloc-96	42	13	71.42%	63.50%	58.86%	100.00%	99.94%	99.96%
83bec2... [39]	UAF	kmalloc-4096	8	8	12.50%	13.70%	9.42%	100.00%	98.16%	97.00%
77e2cf... [38]	UAF	kmalloc-192	21	1	100.00%	95.74%	84.92%	100.00%	100.00%	98.88%
CVE-2017-6074 [29]	DF	kmalloc-2048	16	4	81.25%	80.64%	75.48%	100.00%	100.00%	98.20%
6b8d6b... [37]	DF	kmalloc-512	32	1	100.00%	96.28%	94.92%	100.00%	99.98%	99.90%

Table 2: Exploitation results on real-world vulnerabilities

6.2 Real-World Vulnerability

6.2.1 Real-World Exploitation Setup

In order to prove that PSPRAY is truly effective in exploiting vulnerabilities, we collect 10 real-world vulnerabilities. Among those, seven are from public CVEs, and the remaining three are collected from `syzbot`, which is a list of vulnerabilities found using Syzkaller [40]. In our evaluation, we utilize publicly available exploits for public CVEs. Since the remaining three cases from `syzbot` do not have publicly available exploits, we develop an exploit for them. All these vulnerabilities are ported to Linux `v4.15` and are evaluated when the slab freelist random is enabled. All vulnerability exploitation aims to determine whether the target object can be corrupted.

We evaluate real-world exploitation under two circumstances: 1) idle state and 2) busy state. The idle state is simulated without any other processes being executed. The busy state is simulated by running `stress-ng` to prove the effectiveness of PSPRAY on system workloads. The program, `stress-ng`, spawns two workers for each CPU, IO, and VM, continuously occupying 100% CPU usage on all CPUs. Additionally, to prove that `stress-ng` creates a heavy workload, we measure the number of background heap operations under the two conditions without workload (idle) and with workload (busy) in §A.1.

In this evaluation, each experiment consists of 5,000 rounds, and each round triggers the exploitation one time. This is because one trigger per execution should give more reliable results. Indeed, kernel panics due to the failed exploits (even if the kernel is not crashing) make the kernel unstable, and the vulnerability cannot be reliably triggered again.

6.2.2 Real-World Exploitation Results

Table 2 shows the result of the exploitation of real-world vulnerabilities under two circumstances (i.e., idle and busy). In our evaluation, all success rate of vulnerabilities using PSPRAY outperform general heap spraying exploitation techniques.

Overall, the measured success rate of heap spraying is lower than we expected (i.e., $P_{\text{OOB}}^{Baseline}$). This is because we do not account for the probability of the noise. Noise can occur when spraying the target object. That is, the other process can allocate an object during the allocation of target objects, which deviates from the theory that only target objects are allocated sequentially. Noise occurs more frequently in the busy state than in the idle state. Therefore, the success rate of the idle state is higher than the success rate of the busy state.

More specifically, for the case of OOB vulnerabilities, the success rate on average is 29.48% under an idle state and 22.19% under a busy state with heap spraying. The vulnerability that shows the lowest success rate is CVE-2017-7184, which shows a 21.18% success rate in idle state and 18.86% in busy state. This is because noise can occur while the target object is being sprayed and right before the vulnerable object allocation.

However, the exploitation success rate using PSPRAY appears to be similar to the success rate we expected (i.e., $P_{\text{OOB}}^{\text{PSPRAY}}$). The case where the success rate has increased the most is CVE-2017-7184. It increased by about 75.34% under idle state and 73% under busy state. The reason for this is that after allocating target objects and one vulnerable object, we allocate the one sprayed object to verify that the slow-path has been executed. Through this method, if slow-path is executed, it is possible to indirectly check whether the vulnerable object and target objects are allocated within one slab. That is, this method can indirectly notice the noise occurrence (i.e., other object allocation). For example, if noise occurs so that other object is allocated, allocating one sprayed object does not trigger slow-path. However, there is an exception if the noise allocates the objects as much as the number of objects per slab. At this time, allocating one sprayed object triggers slow-path, but it is hard to guarantee that one vulnerable object and N-1 target object are allocated in one slab. We suspect this is the reason why the workload does not significantly impact the success rate of OOB. However, since it is still possible that the vulnerable object is allocated to the last local object in the slab, the success rate does not reach

100%.

In the case of UAF and DF vulnerabilities, the success rate of the general exploitation method is also slightly less than we expected (i.e., $P^{Baseline}_{\text{UAF \& DF}}$). This is because the noise that executes the context switch can occur between freeing the vulnerable object and allocating the target object. In addition, in the busy state, noise occurs more often than in the idle state. More specifically, among UAF and DF vulnerabilities, the vulnerability that shows the lowest success rate is 83bec2 with 13.70% under idle state and 9.42% under busy state with heap spray. This is because 83bec2 allocates the vulnerable object and seven additional objects. However, the number of objects per `kmalloc-4096` slab is eight. In other words, for the exploitation to succeed by bypassing the change in CPU's `page` resulting from additional object allocation, it needs a new slab that has not been used before.

However, using PSPRAY, the success rate in the idle state shows a nearly perfect success rate. This is because the time window (i.e., after the allocation of the vulnerable object and before the allocation of the target object) in which noise should occur is very narrow in UAF and DF compared to OOB. Therefore, even in the busy state, noise does not greatly affect the success rate. This is why the average success rate under the busy state (i.e., 98.98%) is not significantly different from the average success rate under the idle state (i.e., 99.68%).

7 Mitigation

This section proposes a new defense mechanism against PSPRAY. Recall that PSPRAY is a timing side-channel attack that takes advantage of the following two factors: i) a slow-path allocation is significantly slower than other allocation paths, which can be noticed through timing-channel attacks; ii) when the slow-path allocation takes place, the attacker can learn the allocation status of `freelist`—i.e., an empty `freelist`.

Understanding aforementioned factors, one may consider to employ one of two following methods:

- Ensure uniform allocation performance for all allocation paths, eradicating the first factor.
- Randomize the slow-path allocation context, eradicating the second factor.

Mitigation #1. Uniform Allocation Performance. The first method's main advantage is that the attacker cannot distinguish which allocation path has been taken per allocation request because every allocation would take the same time. To achieve this, the kernel developer may insert dummy code (e.g., a meaningless loop) into fast- and medium allocation paths, which evens out all the allocation performances into worst cases. However, the disadvantage is that the performance degradation would be considerable. More importantly, the SLUB allocator applies several optimization techniques to improve performance. Thus, in order to apply this approach, the SLUB allocator would need to disable all these optimizations. To summarize the first mitigation method, although its design can be fairly simple and effective, we do not think it is an appropriate mitigation against PSPRAY, considering the criticality of the allocation performance in kernel.

Algorithm 1: Randomized slow-path allocation context

```
1  Input C: information about kmem_cache
2  Input Slow_path_index: a public constant index to trigger slow-path
3  Function slab_alloc_node(C):
4      if !is_freelist_empty(C) then
5          Obj = fast_path(C);
6      else if !is_cpu_page_empty(C) then
7          Obj = medium_path1(C);
8      else if !is_cpu_partial_empty(C) then
9          Obj = medium_path2(C);
10     else if !is_node_partial_empty(C) then
11         Obj = medium_path3(C);
12     else
13         Obj = slow_path(C);
14     if Slow_path_index == index_within_slab(Obj) then
15         if is_CPU_partial_empty(C) then
16             new_slow_path();
17     return Obj
```

Mitigation #2. Randomized Slow-path Allocation Context. The second method is to randomize the slow-path allocation context, which is independent of the allocation status of `freelist`. Note that PSPRAY abuses the fact that once the slow-path allocation takes place, the `freelist` is always empty. More technically, when both page and partial are empty, an empty page is immediately allocated to the page through the slow-path, allowing the attacker to learn the `freelist` status. In order to mitigate this, one can randomize the invocation context of the slow-path, thereby ensuring that the slow-path does not always result in an empty `freelist`. In other words, even if the attacker notices the slow-path, the freelist can be in any allocation status—i.e., the `freelist` may be completely empty, or any number of objects were allocated within the freelist.

In order to randomize the slow-path allocation context, one can leverage the existing randomness of allocation order, which can effectively simplify its implementation. Specifically, one first determines the constant object index within the range of a slab, which we call a slow-path index—i.e., from zero to N-1, where N is the number of objects in the slab. This slow-path index is a public constant, which is secure even if known to the attacker. This is because the object allocation over the slow-path index would take place at random as the slab freelist is randomized. Next, when the allocation is performed over the slow-path index, the slow-path is taken and

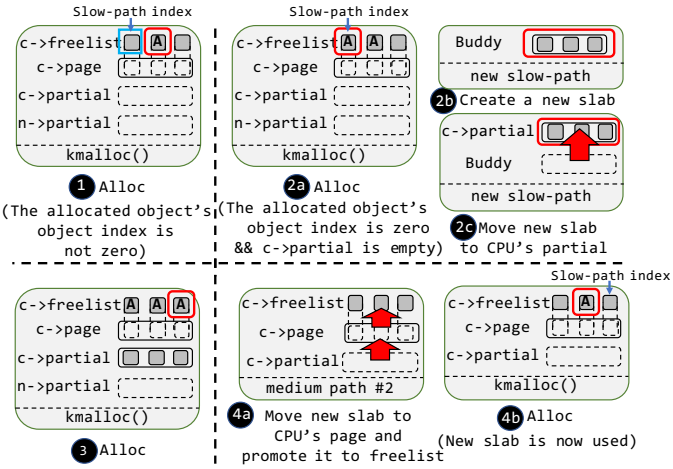


Figure 12: Runtime snapshots of mitigation with the randomized slow-path allocation context.

thus the partial would be filled up.

This second method prevents the attacker from learning the allocation status. This is because even after the slow-path, the `freelist` can be in any allocation status due to the randomness of the slow-path index. In terms of performances, the second method would not introduce much overhead in runtime speed and memory uses. It may allocate the page slightly ahead of time, which should have been allocated without this mitigation. According to our detailed performance evaluation results §A.3, the average overhead is around 1%.

To summarize the second method, although its design may be more complicated than the first, it is a reasonable mitigation method against PSPRAY as it incurs negligible performance overheads.

Implementation: Randomized Slow-path Allocation Context. The algorithm of randomized slow-path allocation contexts is illustrated in Algorithm 1. The modification over the stock kernel is fairly simple, which is highlighted in red color (line 14-16). In our actual implementation with the stock Linux kernel (`v4.15`), the mitigation only modifies 13 lines of code. In order to provide the randomness to the slow-path index, we decided to use slab freelist random, which randomizes the allocation sequence of objects. In our current implementation, we used a value `zero` as the slow-path index (line 14). Then we modified the kernel function, `slab_alloc_node`. When the zero index is used for the allocation, the kernel checks whether the CPU's `partial` is empty (line 15). If it is empty, the kernel creates a new slab with the buddy allocator and assigns the new slab to the CPU's `partial` (line 16).

Figure 12 illustrates an example of our mitigation, showing runtime snapshots when four objects are allocated in order. When the first object is allocated, the slow-path is not triggered as its object index is not zero (❶). When the second object is allocated, now the mitigation takes effect as its allocation takes place over the zero index (❷a). It checks if node's `partial` is empty, and if so, the kernel takes the slow-path. Within this slow-path, the kernel creates a new slab with the buddy allocator (❷b) and assigns the new slab to the node's `partial` (❷c). After that, the kernel allocates the third objects, which uses up all objects in the current CPU's `page` (❸). As a result, this would make the CPU's `page` empty. Next, when the kernel allocates the fourth object, the kernel notices that the CPU's `page` is empty, so it promotes new slab from the partial to the page through medium-path #2 (❹a). Lastly, the kernel allocates the object from the new slab (❹b), completing all the allocation for four objects.

Revisiting the security assurance of this mitigation method, the slow-path can be triggered at any point depending on how the `freelist` is randomized. In the case of this example, the slow-path is triggered when the second object is allocated. However, since each slab has its own random allocation order of freelist, the slow-path would be randomly triggered, stopping PSPRAY's timing-channel attacks.

Security Evaluation. To confirm the security effectiveness of our mitigation, we performed object allocation on `kmalloc-2048` using `msgsnd()` in the same way that we did with PSPRAY. Figure A.6 shows the performance difference between a new slab through the new slow-path and an object through a fast-path when allocating. This figure shows that unlike PSPRAY, where the slow-path is executed periodically in 16 units (i.e., the number of objects per slab), the new slow-path is executed irregularly.

Table A.4 shows the result of the exploitation of a real-world vulnerability. The overall success rate of using PSPRAY under mitigation becomes similar to the success rate without using PSPRAY. This is because, even if PSPRAY finds the timing when the new slow-path is executed, it does not mean the new slab would be used immediately. Therefore, finding a new slow-path is meaningless when exploiting the heap-based vulnerabilities under the proposed mitigation.

Overhead Evaluation. Due to the space limit, we present the detail of performance overheads and memory overheads in §A.3.

8 Discussion

8.1 The Noise

Even if we use PSPRAY, the noise stemming from the scheduler can hinder the exploitation. This is why the probability of success we obtained through the probability model is slightly higher (1.5% higher for OOB and 0.5% higher for UAF and DF) than the actual probability of success through evaluation. The noise can occur from two mechanisms: the CPU migration and context switch. These noises can occur while allocating a vulnerable object or a target object when exploiting the OOB vulnerability. Also, the noises occur between freeing a vulnerable object and allocating a target object when exploiting the UAF and DF vulnerabilities.

CPU migration. If the CPU migration occurs, the CPU's page is changed to another one of the CPU's page so that it hinders the allocator to place the two objects adjacent or to place two objects at the same address. However, it can be prevented using the Linux system call, `pthread_setaffinity_np()`. This system call pins the process, which limits it to operate only on the CPU set. In other words, using CPU pinning, we can mitigate the noise caused by the CPU migration.

Context switch. The context switch can allocate an unexpected object, which is allocated by other processes hindering the placement of a vulnerable object and target object adjacent to each other when exploiting the OOB vulnerability. Also, an unexpected object can be placed at the same address as the vulnerable object, that is, the target object is interrupted to place at the same address as the vulnerable object. It can be alleviated through higher scheduling priority.

However, despite the suppression, context switching does happen sometimes. To address this context switching's side effects, PSPRAY only attempts to exploit heap vulnerabilities when the slow-path is executed three times in a row for a specific period (i.e., the number of objects in one slab), and this number can be increased to further increase accuracy. PSPRAY only does this because this expected period (i.e., three times in a row for this period) can change when other processes allocate and free the object to the target cache. More specifically, if other processes allocate the object to the target cache, the slow-path is executed faster than the expected period. If another process frees the object to the target cache, on the other hand, the kernel will execute the slow-path later than expected. However, it is still possible to fail if the context switching occurs after PSPRAY-ing and before allocating a vulnerable object. This is because there is currently no reliable way to find out whether the context switch has occurred in a very short time period (i.e., between PSPRAY-ing and allocating a vulnerable object).

8.2 The other OSes

To demonstrate the effectiveness of PSPRAY for other OSes, we measure the applicability of PSPRAY to each kernel allocator. Overall, we confirm that the kernel heap allocator of other OSes is similar to that of the Linux kernel SLUB allocator.

FreeBSD. FreeBSD is a free and open-source Unix-like operating system. In FreeBSD, the kernel often uses `malloc()` for kernel heap allocation. `malloc()` allocates the object to the `malloc-X` zone, where `X` is the size of the object. Also, when allocating the new object, the kernel seeks the current `bucket`. If the `bucket`'s count is zero, the kernel calls `cache_alloc()` to find an extra `bucket` to use. If there is none, the kernel calls `zone_alloc_bucket()`, which creates a new `bucket`.

Figure A.7 shows the code for measuring the performance of `malloc()`. We allocate the 1,000 objects with 512 sizes, which are allocated in `malloc-512` zone. `malloc-512` zone consists of a number of pages, each sized at 0x1000. That is, eight objects can be placed per page. Figure A.8 shows the result of measuring the performance of `malloc()` from trial 100 to 200. It can be seen that the performance increases for each of the eight allocations. In other words, this figure shows that a new page is allocated within eight units.

XNU. XNU is the computer operating system kernel developed at Apple. XNU often uses `IOMalloc()` for kernel heap allocation. It allocates an object to the `kalloc.X` zone. More specifically, `IOMalloc()` calls `zalloc_item_slow()` when it is necessary to refill the zone. Otherwise, it calls `zalloc_item_fast()` if the zone has enough free elements. In `zalloc_item_slow()`, it expands the zone, that is, creating a new page.

Figure A.9 is the code of measuring the performance of `IOMalloc()`. We allocate the 1,000 objects with 256 sizes, which are allocated in `kalloc.256` zone. `kalloc.256` zone consists of a number of pages, each sized at 0x1000. Therefore, a total of 16 objects can be placed per page. Figure A.10 shows the result of measuring the performance of `IOMalloc()` from trial 200 to 300. It can be seen that the performance increases noticeably for every 16 allocations. In other words, PSPRAY can be also possible on XNU (Mac OS kernel).

9 Related work

Kernel Automated Exploit Generation. FUZE [41] finds a spot where a new Use-After-Free occurs through fuzzing and employs the symbolic execution to escalate the exploitability. KOOBE [5] extracts the capabilities of a slab-out-of-bound access vulnerability and then automates the process of exploitation. These works focus on finding the exploitability of a vulnerability, so if we combine existing works and PSPRAY, we can discover more vulnerabilities that are exploitable with high exploitation reliability.

Kernel Exploit Techniques. ret2usr [18] and ret2dir [19] are kernel exploitation techniques that modify the control flow to userspace or direct mapped memory. Kepler [42] transforms a control-flow hijacking primitive into single-shot exploitation chain. Eloise [6] uses the elastic object to bypass the KASLR and heap cookie protector. ExpRace [22] increases the time window using the inter-process and hardware interrupt. Existing studies have focused on how to link target objects to RIP control or information leakage when the target objects are corrupted. Conversely, PSPRAY focuses on how to better corrupt target objects. Therefore, it would be a further improvement for exploiting vulnerabilities to combine PSPRAY and these previous works.

Timing Side-Channel Attack against Kernel. Many works [15–17, 21, 24] are trying to circumvent the KASLR using timing side-channel attacks. For example, Hund *et al.* [16] presents a timing side channel attack against KASLR. It focuses on the timing difference caused by the OS page fault handler. Alternatively, Drk [17] measures the timing differ-

ence using the Intel TSX abort handler to reduce the noise. Further, Meltdown [24] applies the speculative execution to place the value of kernel memory on the cache and uses timing difference to read the corresponding value. These works focus on bypassing the KASLR, but PSPRAY focuses on increasing the exploitation reliability of memory corruption that occurs in the kernel heap area.

10 Conclusion

Exploiting Linux kernel heap vulnerabilities is exceptionally difficult due to the operating principle of the default Linux SLUB allocator. In this paper, we analyze these exploitation failures that occur in the kernel heap area depending on the operating principle of the default Linux SLUB allocator. Then, we develop PSPRAY, a new heap spraying technique that uses a timing side-channel, which can significantly increase the probability of an attacker's exploit. Through synthetic and real-world vulnerabilities, PSPRAY demonstrates that it can circumvent the low exploitation success rates issue (e.g., the success rate of 83bec2 from 13.70% to 98.16%). Furthermore, before this problem is abused by attackers, we suggest a mitigation technique for PSPRAY that successfully conceals when a new slab is used with negligible performance and memory overhead (around 1%).

11 Acknowledgment

This work was partially supported by Supreme Prosecutor's Office of the Republic of Korea grant funded by Ministry of Science and ICT (No.1275000160) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone), and National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00745, The Development of Ransomware Attack Source Identification and Analysis Technology).

References

[1] Config_slab_freelist_random: Randomize slab freelist. https://cateee.net/lkddb/web-lkddb/SLAB_FREELIST_RANDOM.html.

[2] Spec cpu 2017. https://www.spec.org/cpu2017/.

[3] Systemd-analyze. https://www.freedesktop.org/software/systemd/man/systemd-analyze.html.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Boston, MA, June–July 2005.

[5] W. Chen, X. Zou, G. Li, and Z. Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (Security)*, BOSTON, MA, Aug. 2020.

[6] Y. Chen, Z. Lin, and X. Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Nov. 2020.

[7] corbet. The slub allocator. https://lwn.net/Articles/229984/.

[8] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[9] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[10] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel {TSX}. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[11] J. Edge. Kernel address space layout randomization, 2013. https://lwn.net/Articles/569635/.

[12] J. Edge. Control-flow integrity for the kernel, 2020. https://lwn.net/Articles/810077/.

[13] C. Evans. What is a "good" memory corruption vulnerability?, 2015. https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.

[14] D. J. Finney. The fisher-yates test of significance in 2× 2 contingency tables. *Biometrika*, 35(1/2):145–156, 1948.

[15] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[16] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[17] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[18] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[19] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2019.

[21] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroSP)*, online, Sept. 2020.

[22] Y. Lee, C. Min, and B. Lee. Exprace: Exploiting kernel races through raising interrupts. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.

[23] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. {PAC} it up: Towards pointer integrity using {ARM}

pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, SANTA CLARA, CA, Aug. 2019.

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, BALTIMORE, MD, Aug. 2018.

[25] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[26] L. W. McVoy, C. Staelin, et al. lmbench: Portable tools for performance analysis.

[27] MITRE. CVE-2010-2959., 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2959.

[28] MITRE. CVE-2016-6187., 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6187.

[29] MITRE. CVE-2017-6074., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6074.

[30] MITRE. CVE-2017-7184., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7184.

[31] MITRE. CVE-2017-7533., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533.

[32] MITRE. CVE-2018-6555., 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6555.

[33] MITRE. CVE-2019-2215., 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2215.

[34] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[35] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium (Security)*, Montreal, Canada, Aug. 2009.

[36] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[37] Syzkaller. Syzkaller log 77e2cfee3bc0fdd3bcaf05ea83a9c26a59ddbf6c., 2018. https://syzkaller.appspot.com/bug?id=77e2cfee3bc0fdd3bcaf05ea83a9c26a59ddbf6c.

[38] Syzkaller. Syzkaller log 6b8d6b1847122db76e4ebd32b9d580684bac133c., 2018. https://syzkaller.appspot.com/bug?id=6b8d6b1847122db76e4ebd32b9d580684bac133c.

[39] Syzkaller. Syzkaller log 83bec290888c08680fb630ec3a2bc87d0ad4b73f., 2018. https://syzkaller.appspot.com/bug?id=83bec290888c08680fb630ec3a2bc87d0ad4b73f.

[40] D. Vyukov. Syzkaller, 2015. https://github.com/google/syzkaller.

[41] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, BALTIMORE, MD, Aug. 2018.

[42] W. Wu, Y. Chen, X. Xing, and W. Zou. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, SANTA CLARA, CA, Aug. 2019.

[43] Y. Yarom and K. Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

A Appendix

A.1 Heavy workload test with `stress-ng`

Background heap operation. To demonstrate that `stress-ng` creates a heavy workload, we measured the number of the background heap operations (i.e., `kmalloc()` and `kfree()`) under an idle state (i.e., without workload) and a busy state (i.e., with workload). Table A.2 shows the number of background heap operations under the two conditions, without workload and with the workload, respectively. To perform this experiment, we modified the kernel to count the number of heap operations in each cache. We then measured the number of heap operations in 100 seconds. As a result, in an idle state, the kernel allocates and frees around 42 objects per second. On the other hand, in a busy state, the kernel allocates and frees around 46,102 objects per second. We think such a difference would clearly sets up the heavy workload environment.

Pspray-ing. Since `stress-ng` allocates many objects, the PSPRAY's process of finding when the slow-path is executed also slows down. More specifically, PSPRAY can determine this moment only when the slow-path is executed three times in a row for a specific period as described in §8.1. If `stress-ng` allocates a number of objects in a short period, the slow-path is not executed for a specific period. Table A.3 shows the time that PSPRAY took to execute. Overall, in all cache, it can be seen that under workload is slower than without workload. However, PSPRAY succeeds in an average of about one second even though the speed is slowed down with workload. We think such a second delay should not be problematic when launching the exploitation attack.

A.2 Success rate according to the number of dummy objects

To find out the effect of the number of additional dummy objects on the UAF and DF expoit success rate, we conduct additional experiments. Figure A.5 shows the success rate with a fixed number of dummy objects. Overall, the more additional dummy objects included, the lower the success rate becomes. The sharp drop in the success rate occurs with the slab with eight objects (i.e., kmalloc-4096). Then, the success rate does not change significantly from more than a certain number of objects (e.g., after 7 additional objects in the red line and after 15 additional objects in the green line). This is because even if the number of objects to be allocated increases when the last object fills the slab, the slab is moved to the full list, and the CPU's `page` is emptied. At this time, the slab that contains the first freed object is assigned to the CPU's `page`. Note that, the average number of additional dummy objects in our real-world evaluation (i.e., based on six UAF and DF vulnerabilities in Table 2) is four.

A.3 Performance Overhead of Mitigation

This section evaluates the performance and memory overhead of our proposed mitigation. The experiment was performed on Linux v4.15, using the same configuration as we evaluated PSPRAY. We measured the performance and memory usage in three cases: 1) after system booting, 2) running LMBench, and 3) running SPEC CPU2017.

Measurement Method: Performance Overhead. To measure the performance overhead, we use three different performance benchmark tools. The first benchmark tool is systemd-analyze [3], which measures the system boot-up performance. The second benchmark tool is LMBench v3.0 [26], which measures the latency and bandwidth of common system calls and I/O operations. The third benchmark tool is SPEC CPU2017 [2], which contains compute-intensive programs for measuring performance.

Measurement Method: Memory Overhead. To measure the memory overhead of our mitigation, we modified the kernel (i.e., `new_slab()`) to log the number of newly created slabs and the total size of the created slabs and check the total amount of used memory using `/proc/meminfo`.

Result Summary. The average mitigation overhead is around 1% for both performance and memory overheads. We think the performance overhead of mitigation is low because it does not introduce much operational differences. Anyway the slow-path should be executed eventually, but the difference is

that the mitigation performs the slow-path in advance. Moreover, we think the memory overhead is low because the new slow-path only assigns up to one slab for each slab cache in advance. In our evaluation setup, the total number of slab cache is 148. This implies that in the worse case, 148 slabs (i.e., at most 600KB) can be assigned in advance.

In the following, we elaborate evaluation details for each case.

A.3.1 After system booting

Performance Overhead. The tool we used to find the boot time is systemd-analyze. Without mitigation, the kernel spent 1.924s to finish the loading of the kernel and user application. On the other hand, the kernel that applies our mitigation spent 1.925s to finish the loading. Overall, the difference between having mitigation and not having mitigation is 0.05%. In other words, our mitigation does not significantly affect the booting time.

Memory Overhead. We measure the memory usage after system booting. During system booting, the kernel creates 4,686 new slab and assigns 1,840 KB slab without our proposed mitigation. In addition, the total size of all the pages used by kernel is 76,548 KB. On the other hand, when the kernel applies our mitigation, the kernel creates 4,700 new slabs and assigns 1,854 KB for all the slabs. The total size of all the pages the kernel uses with our mitigation is 76,972 KB.

A.3.2 LMBench

Performance Overhead. We evaluate performance overhead with and without our mitigation using LMBench. The test where the most overhead occurred is `fork()+exit()`, and about 1.3% overhead occurred. This is because `fork()` makes a large amount causes of allocation create an almost exact duplicate of the process that calls it. The test where the least overhead occurred is `stat()`, about -0.2% overhead occurred. This is because `stat()` only retrieves the file's information without object allocation. In other words, we think the result is not affected by our mitigation but affected by noise (e.g., interrupt). Overall, the average of ten tests shows 0.655% overhead occurred.

Memory Overhead. We measured the memory usage before and after LMBench is executed, then calculate the difference. While LMbench is executed, the kernel creates 1,776,577 new slabs and assigns 1,137 MB all the slabs without our mitigation. Furthermore, the total page of kernel uses during LMBench is 1,351 MB. On the other hand, when the kernel applies our mitigation, the kernel creates 1,775,661 new slab and assigns 1,141 MB for all the slabs. The total size of the page the kernel uses during LMBench is 1,354 MB. Overall, the average memory overhead is 0.34%.

A.3.3 SPEC CPU2017

Performance Overhead. Using SPEC CPU2017, we measure the performance overhead. The least optimal case where the most overhead occurred is `625.x264`, about 0.31% overhead occurred. This is because `625.x264` encodes video streams that involve many object allocations. In addition, there are 10 tests that do not cause overhead. Overall, the average of all 20 tests in SPEC CPU2017 shows 0.066% negligible overhead.

Memory Overhead. We measure the memory usage before and after SPEC CPU2017 is executed. Without our mitigation, the kernel creates 8,114 new slabs and assigns 3,196 KB for all the slabs and total page of all the pages the kernel uses is 4,243 KB. On the other hand, the kernel which applies our mitigation creates 8,152 new slabs and 3,213 KB is assigned for all the slabs. Furthermore, the total size of all the page the kernel uses during SPEC CPU2017 is 4,284 KB. Overall, the average memory overhead is 0.65%.

Cache	Syscalls
kmalloc-32	sys_setsockopt, sys_keyctl
kmalloc-64	sys_msgsnd, sys_setsockopt
kmalloc-128	sys_msgsnd, sys_bind, sys_fchmod, sys_fchown, sys_fsetxattr, sys_setsockopt, sys_flistxattr
kmalloc-192	sys_setsockopt, sys_msgsnd
kmalloc-256	sys_msgsnd, sys_setsockopt, sys_ioctl
kmalloc-512	sys_keyctl, sys_ioctl, sys_setsockopt, sys_msgsnd, sys_ppoll, sys_poll
kmalloc-1024	sys_setsockopt, sys_msgsnd, sys_write, sys_kexec_load, sys_ioctl
kmalloc-2048	sys_write, sys_setsockopt, sys_msgsnd
kmalloc-4096	sys_readv, sys_setsockopt, sys_pread64, sys_read, sys_lseek, sys_msgsnd, sys_preadv
kmalloc-8192	sys_setsockopt, sys_ioctl, sys_ioperm

Table A.1: The system calls which allocate just one object.

```c
static __inline__ unsigned long long rdtsc(void)
{
  unsigned hi, lo;
  __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
  return ((unsigned long long)lo)|(((unsigned long long)hi)<<32);
}

int main()
{
    int msqid[1000];

    for(int i = 0; i < 1000; i++)
    {
        msqid[i] = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
    }

    for(int i = 0; i < 1000; i++)
    {
        int size = 4096-0x100;
        // int size = 2049-0x100;
        // int size = 1024-0x100;
        struct {
            long mtype;
            char mtext[size];
        } msg;
        memset(msg.mtext, 0, size-1);
        msg.mtype = 1;

        long long int st, des;
        st = rdtsc();
        msgsnd(msqid[i], &msg, sizeof(msg.mtext), 0);
        des = rdtsc();

        printf("%d-%lld\n", i, des - st);
    }
}
```

Figure A.1: Proof-Of-Concept of PSPRAY using `msgsnd()`

Cache	w/o workload	w/ workload	Ovehead %
kmalloc-32	4.67	33.63	620.12%
kmalloc-64	8.82	15.77	78.79%
kmalloc-96	0.51	45.16	8754.90%
kmalloc-128	3.47	11455.30	330023.91%
kmalloc-192	9.64	11458.78	118767.01%
kmalloc-256	4.28	81.96	1814.95%
kmalloc-512	0.89	44.10	4855.05%
kmalloc-1024	3.06	22910.59	748612.09%
kmalloc-2048	0.17	12.01	6964.70%
kmalloc-4096	7.10	45.41	539.57%
Total	42.61	46102.71	108096.92%

Table A.2: The number of background heap operations each second on each CPU under w/o workload (idle) and w/ workload (busy).

Cache	w/o workload (s)	w/ workload (s)	Delayed (s)
kmalloc-32	0.1700	0.3313	0.1613
kmalloc-64	0.2083	0.5603	0.3520
kmalloc-96	0.1657	0.6598	0.4941
kmalloc-128	0.1293	0.4587	0.3294
kmalloc-192	0.2328	1.0539	0.8211
kmalloc-256	0.2320	0.6929	0.4609
kmalloc-512	0.2157	0.7770	0.5613
kmalloc-1024	0.2319	1.6589	1.4270
kmalloc-2048	0.1652	1.4823	1.3171
kmalloc-4096	0.1352	1.9823	1.8471
Average	0.1886	0.9657	0.7771

Table A.3: The time cost to success PSPRAY under w/o workload and w/ workload.

```c
char *oob_ptr;
SYSCALL_DEFINE2(oob_test, int, size, int, nonce)
{
        // ALLOC
        oob_ptr = kzalloc(size, GFP_KERNEL);

        int *p = (int *)(oob_ptr + size);
        // OOB READ
        int v = p[12];

        if(v == nonce)
        {
                kfree(oob_ptr);
                return 0x1337;
        }

        kfree(oob_ptr);
        return -1;
}
```

Figure A.2: Synthetic Out-Of-Bounds read vulnerability code

```c
int *uaf_ptr;
int *dummy_ptr[4];
int *ptr;

SYSCALL_DEFINE2(uaf_test, int, cmd, int, size)
{
        if(cmd == 1)
        {
                // ALLOC
                uaf_ptr = kzalloc(size, GFP_KERNEL);
        dummy_ptr[0] = kzalloc(size, GFP_KERNEL);
                dummy_ptr[1] = kzalloc(size, GFP_KERNEL);
                dummy_ptr[2] = kzalloc(size, GFP_KERNEL);
                dummy_ptr[3] = kzalloc(size, GFP_KERNEL);
        }
        else if(cmd == 2)
        {
                // FREE
                kfree(uaf_ptr);
        }
        else if(cmd == 3)
        {
                // USE
                int v = uaf_ptr[12];

                if(v == 0xDEADBEEF)
                {
                        return 0x1337;
                }
                return -1;
        }

        return -1;
}
```

Figure A.3: Synthetic Use-After-Free read vulnerability code

```c
int *df_ptr;
int *dummy_ptr[4];
int *ptr;

SYSCALL_DEFINE2(df_test, int, cmd, int, size)
{
        if(cmd == 1)
        {
                // ALLOC
                df_ptr = kzalloc(size, GFP_KERNEL);
        dummy_ptr[0] = kzalloc(size, GFP_KERNEL);
                dummy_ptr[1] = kzalloc(size, GFP_KERNEL);
                dummy_ptr[2] = kzalloc(size, GFP_KERNEL);
                dummy_ptr[3] = kzalloc(size, GFP_KERNEL);

        }
        else if(cmd == 2)
        {
                // FREE & DOUBLE FREE
                kfree(df_ptr);
        }

        return -1;
}
```

Figure A.4: Synthetic Double-Free vulnerability code

CVE	Bug Type	Slab Cache	# of obj per slab	# of alloc	$P_{\text{OOB\_Random}}$	Heap Feng Shui	$P^{\text{PSPRAY}}_{\text{OOB\_Random}}$	PSPRAY w/o mitigation	PSPRAY w/ mitigation
CVE-2017-7533 [31]	OOB	kmalloc-256	32	2	48.43%	33.78%	96.87%	94.26%	31.84%
CVE-2017-7184 [30]	OOB	kmalloc-128	32	1	48.43%	21.18%	96.87%	96.52%	23.32%
CVE-2016-6187 [28]	OOB	kmalloc-128	32	1	48.43%	23.38%	96.87%	95.58%	30.14%
CVE-2010-2959 [27]	OOB	kmalloc-512	32	2	48.43%	39.60%	96.87%	94.80%	35.48%
CVE	Bug Type	Slab Cache	# of obj per slab	# of alloc	P_{UAF}	Baseline	$P^{\text{PSPRAY}}_{\text{UAF}}$	PSPRAY w/o mitigation	PSPRAY w/ mitigation
CVE-2019-2215 [33]	UAF	kmalloc-512	32	2	96.87%	93.28%	100.00%	100.00%	93.42%
CVE-2018-6555 [32]	UAF	kmalloc-96	42	13	71.42%	63.50%	100.00%	99.94%	65.34%
83bec2... [39]	UAF	kmalloc-4096	8	8	12.50%	13.70%	100.00%	98.16%	12.84%
77e2cf... [38]	UAF	kmalloc-192	21	1	100.00%	95.74%	100.00%	100.00%	96.12%
CVE-2017-6074 [29]	DF	kmalloc-2048	16	4	81.25%	80.64%	100.00%	100.00%	81.12%
6b8d6b... [37]	DF	kmalloc-512	32	1	100.00%	96.28%	100.00%	99.98%	96.32%

Table A.4: Exploitation result on real-world vulnerability with mitigation

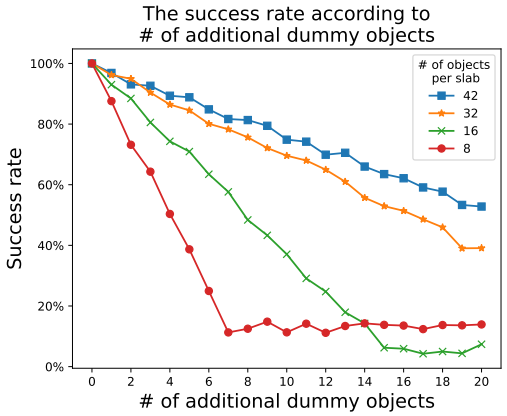


Figure A.5: The success rate according to the number of additional dummy objects when exploiting UAF vulnerability

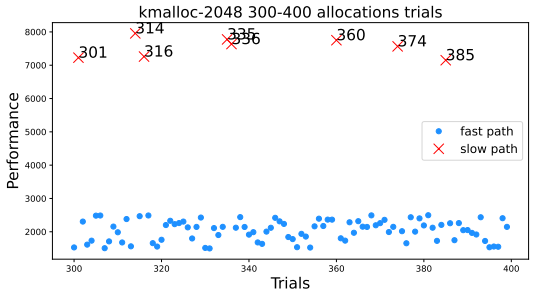


Figure A.6: Measuring the performance of `msgsnd()` when the mitigation is applied

Test	w/o mitigation	w/ mitigation	Overhead
Systemd-Analyze (s)			
Kernel	1.249	1.249	0.00%
User	0.675	0.676	0.15%
Total	1.924	1.925	0.05%
LMbench - latency (ms)			
syscall()	0.5157	0.5174	0.3%
open()/close()	2.6810	2.7021	0.8%
read()	0.7644	0.7725	1%
write()	0.7134	0.7199	0.9%
select() (10 fds)	0.8680	0.8728	0.55%
select() (100 fds)	1.5100	1.5212	0.7%
stat()	1.3038	1.3005	-0.2%
fstat()	0.8282	0.8282	0%
fork() + exit()	115.3949	116.9343	1.3%
fork() + execve()	418.9595	423.8459	1.2%
Average			0.655%
SPEC CPU2017 - latency (s)			
600.perlbench	24.3	24.3	0%
602.gcc	0.006	0.006	0%
603.bwaves	29.1	29.15	0.17%
605.mcf	10.4	10.4	0%
607.cactuBSSN_s	3.45	3.453	0.08%
619.lbm_s	0.93	0.93	0%
620.omnetpp	2.62	2.62	0%
621.wrf_s	6.76	6.77	0.14%
623.xalancbmk	0.045	0.045	0%
625.x264	32.1	32.2	0.31%
627.cam4_s	2.49	2.493	0.12%
628.pop2_s	2.548	2.55	0.07%
631.deepsjeng	8.97	8.98	0.11%
638.imagick_s	0.0205	0.0205	0%
641.leela	4.20	4.20	0%
644.nab_s	2.01	2.012	0.1%
648.exchange2	21.02	21.03	0.04%
649.fotonik3d_s	4.29	4.29	0%
654.roms_s	4.85	4.86	0.20%
657.xz	11.2	11.2	0%
Average			0.066%

Table A.5: The performance of the Linux with and without mitigation.

Test	w/o mitigation	w/ mitigation	Overhead
After System Boot			
# of all created slab	4,686	4,700	0.29%
The memory size of all created slab	1,840 KB	1,854 KB	0.76%
/proc/meminfo	76,548 KB	76,972 KB	0.55%
Average			0.53%
After LMbench			
# of all created slab	1,765,577	1,775,661	0.57%
The memory size of all created slab	1,137 MB	1,141 MB	0.35%
/proc/meminfo	1,351 MB	1,354 MB	0.22%
Average			0.38%
After SPEC CPU2017			
# of all created slab	8,114	8,152	0.46%
The memory size of all created slab	3,196 KB	3,213 KB	0.53%
/proc/meminfo	4,243 KB	4,284 KB	0.96%
Average			0.65%

Table A.6: The memory overhead of the Linux with and without mitigation.

```c
static int lkm_handler(struct module *module, int type, void *arg)
{
    void *p[1000];
    long long int st, end = 0;

    switch(type)
    {
        case MOD_LOAD:
            uprintf("LKM Loaded\n");
            for(int i = 0; i < 1000; i++)
            {
                st = rdtsc();
                p[i] = malloc(512, M_ECHOBUF, M_WAITOK|M_ZERO);
                end = rdtsc();
                uprintf("%d-%lld\n", i, end-st);
            }
            break;
        default:
            retval = EOPNOTSUPP;
            break;
    }
    return retval;
}
```

Figure A.7: Proof-Of-Concept driver source code for FreeBSD

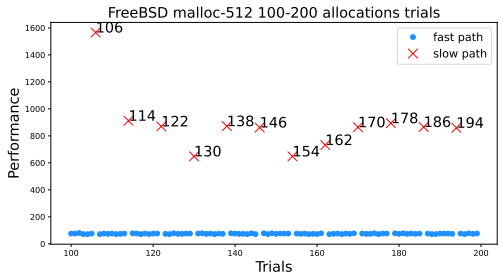


Figure A.8: The result of measuring the performance of `malloc()` in FreeBSD

```c
bool SimpleDriverClassName::start(IOService* provider)
{
    void *p[1000];
    long long int st, end = 0;

    for(int i = 0; i< 1000; i++)
    {
        long long int st = rdtsc();
        p[i] = IOMalloc(256);
        long long int end = rdtsc();
        IOLog("%d-%lld\n", i, end-st);
    }

    return True;
}
```

Figure A.9: Proof-Of-Concept driver source code for XNU

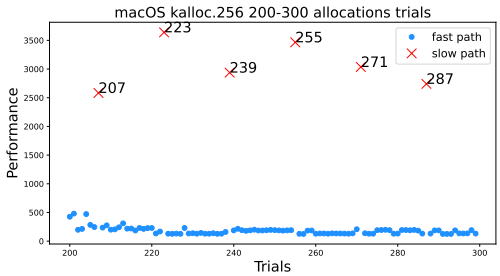


Figure A.10: The result of measuring the performance of `IOMalloc()` in XNU