Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability

Kyle Zeng[*,†], Yueqi Chen[*,‡], Haehyun Cho[†,§],
Xinyu Xing[‡], Adam Doupé[†], Yan Shoshitaishvili[†], Tiffany Bao[†]
[†]Arizona State University, [‡]Pennsylvania State University, [§]Soongsil University
[†]{zengyhkyle, doupe, tbao, yans}@asu.edu, [‡]{ychen, xxing}@ist.psu.edu, [§]haehyun@ssu.ac.kr

Abstract

The dynamic of the Linux kernel heap layout significantly impacts the reliability of kernel heap exploits, making exploitability assessment challenging. Though techniques have been proposed to stabilize exploits in the past, little scientific research has been conducted to evaluate their effectiveness and explore their working conditions.

In this paper, we present a systematic study of the kernel heap exploit reliability problem. We first interview kernel security experts, gathering commonly adopted exploitation stabilization techniques and expert opinions about these techniques. We then evaluate these stabilization techniques on 17 real-world kernel heap exploits. The results indicate that many kernel security experts have incorrect opinions on exploitation stabilization techniques. To help the security community better understand exploitation stabilization, we inspect our experiment results and design a generic kernel heap exploit model. We use the proposed exploit model to interpret the exploitation unreliability issue and analyze why stabilization techniques succeed or fail. We also leverage the model to propose a *new* exploitation technique. Our experiment indicates that the new stabilization technique improves Linux kernel exploit reliability by 14.87% on average. Combining this newly proposed technique with existing stabilization approaches produces a composite stabilization method that achieves a 135.53% exploitation reliability improvement on average, outperforming exploit stabilization by professional security researchers by 67.86%.

1 Introduction

The Linux kernel is an intricate interleaving of many components, working together to power the modern computing landscape. Analogous to user-space software, many of these components dynamically allocate and deallocate memory from a memory region created for this purpose: the "kernel heap". Unfortunately, in the presence of certain programming mistakes, these components can mismanage their allocations, leading to memory errors in the heap that can compromise the security of other components and the whole kernel.

To take advantage of kernel heap memory errors and execute a successful exploit, an attacker must make precise predictions and effect careful control of kernel heap configuration, generally from an attacking process in userspace. However, the intertwined design of relevant Linux kernel components, such as the memory allocator and task scheduler, impacts actual runtime heap layout in complex and unpredictable ways, often making it different from the layout expected by the attacker. This unpredictability of kernel heap layout leads to exploit failures and makes heap-based Linux kernel exploitation notoriously unreliable.

On the one hand, this impact of complexity on kernel heap exploitation reliability seems to help protect kernels from attack. However, on the other hand, it impacts the identification and classification of vulnerabilities as it makes vulnerability reproduction difficult. Overwhelmed developers may deprioritize fixes for vulnerabilities considered to be unexploitable or very unreliable to trigger in favor of other bugs. This gives savvy attackers a window to exploit unfixed vulnerabilities and cause damage to real-world systems.

In this paper, we aim to address this situation by studying and developing techniques that increase the reliability of kernel exploits, helping to properly demonstrate the impact and criticality of the underlying vulnerabilities.

Previous research demonstrates kernel vulnerability exploitability by generating a test case that works with a non-zero probability [21, 30, 42, 43]. On the contrary, little research has been done to address the *exploitation reliability problem* — if a kernel heap vulnerability is exploitable, then how to achieve a high exploitation success rate and properly understand the chance of a successful attack? While kernel heap exploit stabilization techniques, including Heap Grooming [35] and Defragmentation [44], exist in the wild, little progress has been made to systematically determine their efficacy or investigate how to apply these techniques to improve exploit reliability. Instead, human analysts primarily rely upon

[*]First two authors contributed equally to this work.

their personal expertise and intuition in assessing and adopting these techniques, which is ad-hoc and error-prone.

In this work, we present a systematic study on the kernel heap exploit reliability problem. Our study aims to answer the following research questions: (1) What are the common kernel heap exploit stabilization techniques in the wild? (2) What are expert opinions on using these techniques, and what are the real-world evaluation results? Are these expert opinions correct? (3) What are the reasons that a stabilization technique succeeds/fails? (4) How to improve kernel exploit reliability?

To answer these questions, we conducted a hybrid study with both qualitative and quantitative analyses. We interviewed 11 kernel exploitation experts, collecting 5 stabilization techniques along with experts opinions about these techniques. We then implemented exploits for 17 real kernel heap vulnerabilities and measured the reliability improvement associated with each stabilization technique. Finally, we compared the experimental results against the expert opinions and investigated the factors contributing to exploit unreliability.

In our measurements, we found that expert opinions are incorrect in many kernel heap exploit scenarios. For example, many experts considered Defragmentation to improve the reliability of all exploits, but we discovered that Defragmentation is effective only for heap-based out-of-bound access (OOB) vulnerabilities, but not for use-after-free (UAF) and double-free (DF) vulnerabilities. Furthermore, we discovered that the Defragmentation technique actually *reduces* exploit reliability if misused in UAF or DF vulnerability exploitation.

To better understand the kernel heap exploit reliability problem, we devised a generic kernel heap exploit model that illustrates contributing factors and explains why each technique succeeds or fails to improve exploit reliability. Inspired by our model, we created a novel technique, called Context Conservation, to improve exploitation reliability for DF and UAF vulnerabilities. Our evaluation shows that the technique improves exploit reliability by 14.87% on average. When Context Conservation is combined with existing exploit stabilization techniques, this composite method improves exploit reliability by an average of 135.53%. Furthermore, it outperforms the exploits crafted by human experts by 67.86%.

In summary, this paper makes the following contributions:

- A systematic study on the existing kernel heap exploit stabilization techniques commonly used in practice.
- The design of a general kernel heap exploit model that explains the exploit reliability problem.
- The development and evaluation of a sole technique, named Context Conservation, and a technique combination that both significantly improve kernel exploit reliability.

We release the implementation and the experiment dataset[1] to foster future research. We believe the artifacts will benefit the community by helping to stabilize future kernel exploits and to evaluate future exploit stabilization techniques.

[1] https://github.com/sefcom/KHeaps

2 Background

In this section, we introduce the technical backgrounds related to kernel heap exploit reliability problem.

2.1 Exploit-related Linux Kernel Design

Kernel Memory Allocator. Linux kernel uses SLUB allocator for managing the kernel heap. The allocator creates a cache storage unit for kernel objects of the same size. Each cache maintains a list of pages, and each page is called a *slab*. When a fresh slab is created, it is partitioned into multiple *slots* of the same size. The slots are then chained together, forming a *per-slab freelist*. While the per-slab freelist will be shuffled and become randomized after a few heap operations, the layout of a fresh freelist is deterministic.

At the runtime, each CPU operates on different slabs to avoid contention. The freelist of the slab used by a CPU is usually referred to as a per-CPU freelist. When a CPU allocates an object, it takes one slot from its per-CPU freelist for the object. When an object is released, the allocator determines the corresponding slab, and returns the object's slot back to the slab's per-slab freelist.

Task Scheduler. A kernel task scheduler decides the execution order of running processes. Linux kernels use Completely Fair Scheduler (CFS) [1] to schedule processes, which schedules processes in the granularity of threads and calls them `tasks`. It keeps track of all tasks ready to run in a *runqueue*.

In multi-core systems, each CPU maintains its per-CPU runqueue and records the workload in the runqueue. When the workloads on CPUs are imbalanced, the kernel will migrate tasks from a heavy-workload runqueue to a light-workload runqueue. Generally, the busier the system is, the more frequently task migration occurs.

Context Switch. One CPU can execute only one task at a time. A context switch occurs when the task scheduler tells a CPU to stop the current task and run another task to simulate concurrent execution. When it happens to a userspace process, the process will be stopped and wait to be resumed.

Since Linux kernel is preemptive, a context switch can potentially occur at any moment to a running process, which effectively introduces a long-time sleep to the process.

2.2 Exploitation Methods

In general, there are four kernel heap exploit methods:

Out-of-bounds Object (OOB-Object) Exploits. The OOB-Object exploits are resulted from heap object overflow vulnerabilities. The exploits tamper the critical fields in the adjacent object (*i.e.*, target object) and hijack kernel control

flow. For example, it can overwrite function pointers or hijack virtual table pointers adjacent to the overflowed object.

Out-of-bounds Freelist (OOB-Freelist) Exploits. Another way to exploit OOB vulnerabilities is to hijack the slab freelist by overwriting the list metadata stored in the adjacent freed slot (*i.e.*, target slot). This approach drives the kernel to chain an adversary-controlled address into the slab freelist for allocating kernel objects in the desired memory region.

Use-after-free (UAF) Exploits. An UAF exploit occurs in two steps. First, it releases a vulnerable object and leaves the freed slot on the kernel heap using the vulnerability. Then, he/she tricks the kernel to write adversary-controlled data on the freed slot, tampering with critical fields (*e.g.*, pointers, list metadata). Using the same approach as in OOB exploitation, the adversary can obtain control over kernel execution.

Double-free (DF) Exploits. When a double-free vulnerability is triggered, the kernel heap is corrupted with the same slot appearing twice in the freelist, allowing an adversary to hijack the control flow. First, the adversary allocates a victim object at the target slot. Then, he/she can allocate another object with desired data at the same slot, overwriting the victim object. The second allocation overwrites critical fields in the victim object. As a result, the adversary can hijack the program counter in the same approach as described above.

2.3 Exploitation Unreliability Issue

The kernel heap is used at runtime by different kernel components, such as processes, interrupts, softirq, etc. In kernel heap exploitation, these components could impose intensive heap usages, making it unreliable to allocate objects at the desired slot. Even worse, a task migration may occur during exploitation. Due to the per-CPU freelist design, the migration implicitly changes the freelist in use, which can cause the failure of exploitation (§6.1).

Several techniques are commonly used to improve kernel heap exploit reliability in practice. For example, Defragmentation [44] improves exploit reliability by allocating many objects to drain freed slots and force new slab creation. Since the new slab's layout is predictable, the exploit can locate target slots precisely, thus more likely to succeed. In our paper, we call such techniques *kernel heap exploit stabilization techniques*, or *exploit stabilization techniques* for short.

Although the exploit stabilization techniques are prevalent in practice, little research has been done to study these techniques. The security community not only lacks the understanding of why each of the exploit stabilization techniques improves exploit reliability but also fails to mention whether the techniques are effective under all conditions and, if not, what is the condition to apply an exploit stabilization technique. As such, security analysts use these techniques in an ad-hoc way which, as we will show in the later sections, could result in no improvement or even a negative impact on the kernel heap exploit reliability.

3 Overview

Problem Statement. We define the kernel heap exploit reliability problem as the issue that a kernel heap exploit cannot work reliably. While previous works [21, 30, 42, 43] seek for solutions to generate exploits that work with a positive success rate (*e.g.*, an exploit works once out of one hundred attempts), our paper focuses on the problem of how to help these generated exploits gain higher success rates.

Approach. We design a hybrid approach to study the kernel heap exploit reliability problem. Our approach includes both qualitative human subject studies and quantitative experiments. We first conduct a qualitative study on security experts, through which we identify commonly-used kernel heap exploit stabilization techniques and experts' opinions on each technique (§4). Next, we evaluate the stabilization techniques on real-world Linux kernel vulnerabilities and measure the reliability of the corresponding exploits with and without each technique (§5). Based on the experiment results, we identify the experts' misconceptions (§5.3), investigate the factors impactful on exploit reliability (§5.4), and build a generic model to explain the kernel heap exploit reliability problem (§6). Finally, we use the model to explore novel techniques and thus boost exploit reliability (§7).

4 Kernel Heap Exploit Reliability Interview

The interview aims to (1) identify techniques commonly used by experts to improve exploit reliability, (2) collect experts' opinions regarding the techniques, and (3) gather implementation suggestions from experts for later empirical evaluation. As the interview involves human subjects, we submitted our experiment to the IRB and received an exemption.

To recruit participants, we invited security experts to participate in our study and screened the experts as experiment participants by their experience in Linux kernel exploitation. In total, we recruited 11 participants. The participants consist of 4 professional red-team exploit developers and 7 kernel security researchers from academic institutions. All of these participants have more than 2 years of kernel exploit experience. They developed 24 end-to-end Linux kernel exploits on average. We believe this group of experts has adequate experience for our study.

Our interview survey provided a list of exploitation stabilization techniques — we believe — security researchers commonly adopt. As depicted in Table 1, the list contains 5 exploitation stabilization methods. They are summarized from public exploit writeups, broadly available exploit implementation, and tech forums. Following each technique, we also specified different conditions under which the technique may take effect properly. These working conditions were provided to participants for selection as part of the survey. For some techniques, our survey also asks participants to compare

Technique	Empirical Results	#	correctness
Defragmentation	1-1. Works for UAF, DF, and OOB	7	✗
	1-2. Only works for OOB	3	✓
	1-3. Only works when the workload of the target machine is light	4	✗
	1-4. Most effective among all techniques	1	✗
	1-5. Even if this technique does not improve reliability, at least, it doesn't harm reliability	1	✗
	1-6. I don't know this technique	1	N/A
	1-7. I have no clue how to use this technique	0	N/A
Heap Grooming	2-1. Only works for OOB	2	✓
	2-2. Only works when the workload of the target machine is light	3	✗
	2-3. More useful than Defragmentation	1	✓
	2-4. This technique is as good as Defragmentation	1	✗
	2-5. I don't know this technique	7	N/A
	2-6. I have no clue how to use this technique	1	N/A
Single-Thread Heap Spray	3-1. Works for UAF, DF, and OOB	7	✗
	3-2. Most effective among all techniques	1	✗
	3-3. I don't know this technique	1	N/A
	3-4. I have no clue how to use this technique	2	N/A
Multi-Process Heap Spray	4-1. Works for UAF, DF, and OOB	7	✗
	4-2. The single-thread implementation is as good as the multiple-process implementation	1	✗
	4-3. The single-thread implementation is better than the multiple-process implementation	2	✗
	4-4. Most effective among all techniques	1	✗
	4-5. I don't know this technique	1	N/A
	4-6. I have no clue how to use this technique	2	N/A
CPU Pinning	5-1. Effective only when the threads/processes for exploit are pinned to idle CPUs	3	✗
	5-2. Heavy workload will influence the effectiveness	1	✓
	5-3. I don't know this technique	8	N/A
	5-4. I have no clue how to use this technique	0	N/A

Table 1: Summary of security experts' opinions on exploitation stabilization techniques. # column indicates the amount of participants holding the corresponding opinion. In the correctness column, ✗ means the claim is inconsistent with our systematic measurement results whereas ✓ implies consistency.

their effectiveness with each other. The list of open questions in the survey can be found in Figure 2 in the Appendix.

Since the list of exploit stabilization techniques and their working conditions may be limited by our knowledge, the survey is designed to allow the participants to complement the list. Suppose participants have never heard about a specific exploitation method listed in the survey or have no clue about the condition under which the technique could be applied. In that case, the survey also allows them to specify these situations. In Figure 2, we present the full survey questions used in our user study.

Following the survey questions, we interviewed each participant. In the interview, we provided the technical details of 17 real-world kernel vulnerabilities to the participants. As discussed Section 5, these vulnerabilities represent a corpus of test cases that we use to evaluate exploit stabilization techniques. We asked whether the stabilization techniques present in the survey could be applied to improve exploitation reliability for each vulnerability. If so, how they would implement and predict the reliability of the implemented exploit in different workload settings. More specifically, we asked our human subjects the implementation details, including what system calls to use, whether to create separate threads or fork multiple processes for executing the technique code, etc.

Recall that part of our research is to evaluate the effectiveness of the stabilization techniques and then compare these security experts' intuitions with our experiment results. As a result, our user study also includes a post-survey interview. In this interview, we presented and shared our experiment discovery with the participants. We asked whether our experiment discovery helped them better understand a stabilization technique if they did not properly use it. We also asked if our discovery corrected their opinion on an exploitation stabilization technique if their previous answer mismatched with our evaluation result. In this way, we can rule out the possibility that participants' opinion misalignment results from their misunderstanding of our survey questions.

4.1 Stabilization Techniques

Through our survey, we confirmed the following stabilization techniques are those commonly adopted. All the exploit stabilization methods are known by a subgroup of the participants. In the following, we briefly describe these commonly-adopted stabilization techniques.

1. **Defragmentation** allocates a large number of objects in the same cache as the vulnerable object so that it fills up all the half-full slabs in the cache, forcing the allocator to create new slabs.
2. **Heap Grooming** creates a heap layout where the vulnerable object is right next to a victim object. Such layout is beneficial for some of the heap exploits. To achieve the goal, Heap Grooming initializes the kernel heap by allocating many victim objects and then swapping one victim

object with the vulnerable object by performing quick free and malloc operations.

3. **Single-Thread Heap Spray** occupies the desired slot by allocating many objects. Single-Thread Heap Spray is performed in the same thread or a separate thread for better modularization. It is used after triggering a UAF/DF vulnerability or before triggering a OOB vulnerability.
4. **Multi-Process Heap Spray**, similar to Single-Thread Heap Spray, aims to occupy the desired slot for an exploit. The difference lies in the implementation approaches. Single-Thread Heap Spray employs only one thread, whereas Multi-Process Heap Spray forks multiple processes for allocating payload objects.
5. **CPU Pinning** attaches the execution of exploit threads/processes to a specific CPU to prevent task migration. It works by invoking `sched_setaffinity` system call.

4.2 Interview Responses

Table 1 presents the participants' opinions on each stabilization technique. The number in the third column indicates how many participants provide that corresponding response in the survey. Below, we summarized the responses.

- A majority of the experts (roughly half or more than half of the participants) share the same opinion on stabilization techniques like Defragmentation, Single-Thread, and Multi-Process Heap Spray. That is, these techniques could increase the reliability for exploits targeting arbitrary heap-based vulnerabilities. Although most security experts broadly acknowledge this opinion, its correctness lacks systematic evaluation.
- While all the experts have 2 years of experience in developing many end-to-end exploits for Linux kernel vulnerabilities, most are not familiar with the exploitation stabilization techniques like Heap Grooming and CPU Pinning. This finding aligns with our intensive efforts on the Internet. Both techniques are rarely mentioned or adopted in publicly available exploits and writeups, as shown in Table 6 in Section 7.2. It indicates that the knowledge gap exists in the use of some exploitation stabilization techniques. There is a need to unveil the techniques, explore their effectiveness, and summarize the working conditions.
- A small subgroup of security experts holds opposite opinions on some exploitation stabilization techniques. For example, three participants deem Defragmentation takes effect only for OOB vulnerabilities. On the contrary, the majority of the participants believe this stabilization technique should work for any heap-based vulnerabilities. This contradictory opinion indicates that the misunderstandings of these commonly used techniques exist within the kernel exploit experts. A systematic experiment should be conducted to conclude the condition under which the corresponding technique is effective.

Cache	Idle	Busy
kmalloc-8	0.02	0.04
kmalloc-16	0.01	0.43
kmalloc-32	1.13	31716.30
kmalloc-64	0.70	31728.44
kmalloc-96	0.06	1.62
kmalloc-128	0.65	31416.34
kmalloc-192	0.35	11.44
kmalloc-256	1.13	57827.32
kmalloc-512	0.06	26152.33
kmalloc-1024	0.25	41844.99
kmalloc-2048	0.01	10460.07
kmalloc-4096	1.61	21039.32
kmalloc-8192	0.00	0.04

Table 2: The number of background heap operations each second on each CPU under different workload settings.

5 Evaluating Stabilization Techniques

In this section, we first introduce how we evaluate the effectiveness of exploit stabilization techniques. Second, we present our evaluation results. Third, we compare our discovery from evaluation with human experts' opinions on exploitation stabilization techniques. Last but not least, we manually analyze the runtime behavior of each exploit and thus summarize the factors influencing exploitation reliability.

5.1 Experiment Setup

To evaluate the exploit stabilization technique, we set up a comprehensive experiment. First, we collect and construct a dataset that consists of public exploits for real-world vulnerabilities. Second, we craft exploit variants equipped using each stabilization technique for each vulnerability by following the implementation suggestion provided by the interviewed participants. Third, we develop an evaluation platform, run the public exploits and the corresponding variants on that platform, and thus measure the exploitation success rates of each exploit. Last, we compare the success rates and analyze the experiment results. In the following, we detail these experiment setups.

Dataset. We search the Internet and form a dataset containing 17 Linux kernel vulnerabilities and corresponding public exploits. The vulnerabilities in our dataset cover all common vulnerability types on the kernel heap. This dataset is the most extensive corpus of working Linux kernel exploits to the best of our knowledge. As such, we consider it a reasonable dataset for stabilization technique evaluation. For the details of vulnerability/exploit selection, readers could reference Appendix-A.

Exploit Variants. Based on the exploits gathered, we construct baseline exploits for all 17 vulnerabilities. These baseline exploits are not enhanced with any stabilization techniques. From these baseline exploits, we introduce the sta-

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
				In Idle State			
OOB	2017-7533	20.43%	49.03% [+]	N/A	98.31% [+]	99.49% [+]	28.80% [+]
OOB	2017-7184	33.51%	99.46% [+]	100.00% [+]	46.37% [+]	96.74% [+]	52.54% [+]
OOB	2016-6187	32.89%	99.63% [+]	N/A	N/A	N/A	32.71% [−]
OOB	2010-2959	22.09%	99.37% [+]	99.89% [+]	56.63% [+]	57.29% [+]	21.54% [−]
OOB	2017-7308	0.04%	0.32% [+]	N/A	0.00% [−]	5.28% [+]	0.10% [+]
UAF	2018-6555	89.98%	77.54% [−]	N/A	99.68% [+]	100.00%[+]	94.08% [+]
UAF	2016-8655	96.06%	94.28% [−]	N/A	99.46% [+]	99.62% [+]	98.84% [+]
UAF	2017-15649	13.82%	12.52% [−]	N/A	58.54% [+]	99.88% [+]	99.70% [+]
UAF	2016-4557	99.34%	99.03% [−]	N/A	100.00%[+]	100.00%[+]	99.43% [+]
UAF	2017-8824	97.64%	0.10% [−]	N/A	97.92% [+]	99.24% [+]	98.60% [+]
UAF	2016-0728	1.48%	6.64% [+]	N/A	76.10% [+]	99.94% [+]	3.34% [+]
UAF	2017-10661	31.98%	32.16% [+]	N/A	53.76% [+]	84.48% [+]	69.72% [+]
UAF	2016-10150	0.74%*	1.02%* [+]	N/A	1.14%* [+]	99.72% [+]	99.56% [+]
UAF	2017-11176	92.17%	11.86%* [−]	N/A	99.89% [+]	99.91% [+]	97.31% [+]
DF	2017-2636	95.96%	22.58% [−]	N/A	99.62% [+]	99.44% [+]	98.06% [+]
DF	2017-6074	98.50%	84.98% [−]	N/A	98.80% [+]	92.18% [−]	98.98% [+]
DF	2017-8890	51.96%	0.66% [−]	N/A	67.80% [+]	72.82% [+]	81.18% [+]
Statistics (#[+]–#[−]–#[=])			8-9-0	2-0-0	15-1-0	15-1-0	15-2-0
				In Busy State			
OOB	2017-7533	7.86%	23.63% [+]	N/A	74.57% [+]	80.66% [+]	18.23% [+]
OOB	2017-7184	5.37%	25.00% [+]	51.11% [+]	35.80% [+]	73.63% [+]	6.83% [+]
OOB	2016-6187	2.49%	9.89% [+]	N/A	N/A	N/A	4.94% [+]
OOB	2010-2959	5.83%	20.09% [+]	46.60% [+]	18.37% [+]	44.06% [+]	10.03% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.18% [+]	0.00% [=]
UAF	2018-6555	61.92%	61.26% [−]	N/A	81.74% [+]	96.38% [+]	73.66% [+]
UAF	2016-8655	0.28%	0.44% [+]	N/A	59.50% [+]	99.68% [+]	0.34% [+]
UAF	2017-15649	18.34%	16.64% [−]	N/A	37.16% [+]	99.30% [+]	92.02% [+]
UAF	2016-4557	22.80%	17.31% [−]	N/A	94.34% [+]	99.29% [+]	18.20% [−]
UAF	2017-8824	93.70%	0.34% [−]	N/A	95.00% [+]	98.82% [+]	98.66% [+]
UAF	2016-0728	0.14%	0.18% [+]	N/A	33.66% [+]	99.70% [+]	0.06% [−]
UAF	2017-10661	0.18%	0.14% [−]	N/A	28.90% [+]	69.54% [+]	0.24% [+]
UAF	2016-10150	47.62%	46.08% [−]	N/A	39.58% [−]	63.00% [+]	77.70% [+]
UAF	2017-11176	2.06%	1.77% [−]	N/A	40.86% [+]	92.40% [+]	4.03% [+]
DF	2017-2636	93.26%	18.74% [−]	N/A	97.80% [+]	99.20% [+]	98.28% [+]
DF	2017-6074	66.16%	49.80% [−]	N/A	66.84% [+]	75.52% [+]	64.38% [−]
DF	2017-8890	6.14%	1.44% [−]	N/A	20.32% [+]	44.00% [+]	9.24% [+]
Statistics (#[+]–#[−]–#[=])			6-10-1	2-0-0	14-1-1	16-0-0	13-3-1

Table 3: Performance of reliability techniques in idle and busy systems. N/A indicates that the technique cannot be applied to the vulnerability. [+] means improvement over baseline, [−] is for degradation, and [=] denotes no change. Some exploits suffer from timeout issue. For these exploits, we only include exploits that finish execution in 5 minutes for success rate calculation. The numbers may not reflect their real success rates. These exploits are marked with [∗].

bilization technique individually. Due to the space limit, we describe the detail of generating baseline exploits and exploits armed with each stabilization technique in Appendix-A.

Experiment Design. We build an evaluation platform to ease the process of measuring the success rate of exploits. With the platform, we can automatically run all exploits for vulnerabilities and collect the panic logs to determine whether the exploits succeed or not. Moreover, the platform can evaluate exploits under either idle or busy systems to measure system workload's influence on exploit reliability. The busy system is simulated by running Apache benchmark from Phoronix Test Suite [16] on the target system. The benchmark workload spawns more than 10 processes and 150 threads, constantly occupying 81.24% CPU usage on all CPUs and imposing intensive pressure on the kernel heap. The background heap operations (`kmalloc` and `kfree` calls) on idle systems and busy systems are compared in Table 2.

For each exploit, we run it 5000 times and then compute the success rate. In our experiment, we deem an exploit completes its execution if it succeeds or triggers a kernel panic.

This practice follows the convention in the kernel exploit development community [4, 6, 10] and avoids unfair advantages to exploits that include repeated execution logic [8, 9, 11]. We use the success rate of exploits as the indicator of their reliability. We calculate the average value of 5000 times and present them in Table 3.

To perform the evaluation experiment, we run the platform on 4 identical servers, each equipped with Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz (20 cores in total) and 252G memory, running Ubuntu 18.04 LTS. For each vulnerability, we re-introduce it into v4.15 Linux kernel, compile the corresponding kernel and disk image and run the system in `QEMU 5.0.0` virtual machine (VM). The VM is configured in 4 different settings, including 2 CPUs + 2GB RAM, 2 CPUs + 4GB RAM, 4 CPUs + 2GB RAM, and 4 CPUs + 4GB RAM. We did not perform experiments on more CPU/RAM settings for two reasons. First, running experiments with more resources is expensive. Second, we did not observe significant changes in technique effectiveness under different CPU/RAM settings. As a result of the observation, we present only the experiment results on the VM with the configuration of 2 CPUs + 2 GB RAM. For the results obtained from other configurations, readers could refer to Appendix-B.

5.2 Effectiveness Evaluation

Table 3 shows the experiment results. As we can observe, for all vulnerabilities, there is at least one stabilization technique that could successfully improve the reliability of its corresponding exploit. Among all exploit stabilization techniques, Defragmentation demonstrates the weakest capability in improving exploit reliability. It improves exploit reliability only for 8 and 6 vulnerabilities in the idle and busy settings, respectively. This observation indicates that the stabilization techniques commonly used are generally effective.

While Defragmentation is the least effective in terms of the number of exploits it stabilizes, similar to Heap Grooming, Single-Thread Heap Spray, and Multi-Process Heap Spray, it could significantly improve exploit reliability for OOB vulnerabilities. The reason is that these four techniques can reduce the dynamics of the kernel heap and create a clean heap layout for exploitation. In our experiment, no technique dominates the reliability of OOB exploits. While Defragmentation and Heap Grooming show dominant performance in idle systems, their success rates decrease in the busy setting. In contrast, the reliability of Multi-Process Heap Spray stays relatively high even in the busy setting for OOB exploits.

For UAF and DF vulnerabilities, the most effective techniques are Single-Thread Heap Spray and Multi-Process Heap Spray in both idle and busy settings. Take CVE-2016-0728 as an example. The success rate of baseline exploit is 1.48% in idle systems. Single-Thread Heap Spray and Multi-Process Heap Spray can raise the success rate to 76.10% and 99.94%, respectively. Recall that exploiting UAF and DF vulnerabilities requires adversaries to take over the freed slot on the heap. Heap spray allocates many spray objects, increasing the possibility of successful occupation and thus improve reliability. In most cases of our experiment results, we observe that Multi-Process Heap Spray outperforms Single-Thread Heap Spray by a vast margin. This is due to the former's influence on both scheduler and CPU affinity. We will provide a more detailed discussion in Section 6.1.

Interestingly, though Multi-Process Heap Spray shows significant exploit reliability improvement in both idle and busy systems, it negatively impacts one exploit in the idle setting. This hints at other complex underlying factors.

Comparing the success rate of exploits in idle and busy settings, we observe that the workload of the target system does influence the effectiveness of reliability techniques. Take the technique CPU Pinning as an example. In idle setup, 9 out of 17 exploits achieve more than 90% success rate. However, in the busy setup, this number decreases to three. Even so, we still believe the exploit stabilization techniques are effective. This is simply because, even though busy workload degrades exploit reliability in general, each technique's impact on exploits is generally consistent under different workloads.

5.3 Comparison with Expert Opinions

By comparing the results in Table 1 with those in Table 3, we observe that experts have the following incorrect opinions on exploitation stabilization techniques.

Misconception 1. As depicted in Table 1, a majority of participants believe Defragmentation could improve reliability for exploits against arbitrary heap-based vulnerabilities (*i.e.*, UAF, DF, and OOB). However, our experiment result misaligns with their opinions. As is shown in Table 3, in both idle and busy settings, the Defragmentation technique is more effective for OOB vulnerabilities than other types of vulnerabilities like DF and UAF. Among all 17 vulnerabilities, all five OOB exploits benefit from the technique significantly. In contrast, only three out of nine UAF exploits benefit from it. None of the DF exploits with Defragmentation demonstrates improvement. As such, we conclude that the opinions of many experts on Defragmentation are incorrect.

Misconception 2. For almost all techniques, some interview participants believe that heavy workload will significantly harm their effectiveness (*i.e.*, 1-3, 2-2, 5-2 in Table 1). Intuition suggests that increased workload introduces more opponent processes, competing with the exploit threads/processes for CPU time and kernel heap usage. As a result, it is more challenging to perform heap operations in a non-interference fashion.

Our experiment results generally confirm this hypothesis. All techniques suffer success rate degradation in busy systems. However, stabilization techniques still improve the success rate significantly. For example, though Multi-Process Heap Spray in busy systems does not perform as well as in idle

systems, the success rate still stays high for almost all vulnerabilities. This result indicates that *increased workload does influence the effectiveness of techniques but is not a killer*.

Misconception 3. Regarding the question of which technique is the most effective, participants have totally different opinions (*i.e.*, 1-4, 2-3, 2-4, 4-4 in Table 1). In fact, there is *no universal answer* to this question, and the most effective technique varies case by case. Take `CVE-2017-7533` as an example. The most effective technique for the reliability improvement of this technique is Multi-Process Heap Spray, no matter whether the workload is light or heavy. For `CVE-2017-7184` in idle systems, the optimal choice becomes Heap Grooming.

Recall that in Section 4, there are two approaches to implementing heap spray. One is to spray objects in one thread after triggering the vulnerability. The other is to fork multiple processes dedicated to object allocation. Opinions diverge on which implementation is more effective (*i.e.*, 4-2, 4-3 in Table 1). Our experiment result suggests that *Multi-Process Heap Spray outperforms Single-Thread Heap Spray* for almost all cases except `CVE-2017-6074` with a light workload.

Other Misconceptions. The experiment refutes opinions 1-5 in Table 1. We observe that Defragmentation harms exploit reliability for five vulnerabilities. This hints that the outcome is not always as expected by the exploit developers.

Some participants agree that the target CPU should be idle for CPU Pinning to be effective (5-1 in Table 1). This statement is incorrect since CPU Pinning demonstrates substantial reliability improvement in the busy setting, where all CPUs are busy. In our investigation, we observe that the crucial factor of CPU Pinning is that it ensures all exploit processes/threads are all attached to the same CPU. Recall that in Section 2, the allocator has per-CPU feature which allocates/frees the `slot` from/to the local freeslit. Attaching all processes/threads to the same CPU ensures that heap manipulation takes place on the same freelist.

5.4 Summary of Exploit Unreliability Factors

For each vulnerability, we also monitor the runtime behaviors of its different exploits. In this way, we can observe internal kernel operations when running each exploit and thus compare the behavior difference accordingly. From our comparison, a better understanding of the exploitation unreliability could be obtained. Here, we summarize our understanding or, in other words, the factors attributive to exploit unreliability.

Unknown Heap Layout. Before the execution of an exploit, other processes may shuffle the freelist in each slab and destroy their linear layouts. As a result, when the exploit starts its execution, it faces an unpredictable heap layout.

Unknown heap layout has a detrimental effect on the reliability of OOB exploits. As briefly discussed in Section 2.2, OOB exploits aim to allocate a victim object next to a vulnerable object for exploitation. Without knowing the state of the freelist in use, it is difficult to obtain the desired heap layout useful for successful exploitation.

In the typical exploit development process, security researchers usually assume an initial heap layout and develop exploits based on the assumption. If the heap layout does not match the assumption during runtime, the exploit is likely to fail. This assumption on the initial heap layout introduces randomness to exploit reliability.

Unwanted Task Migration. Although seemingly harmless, unexpected task migration contributes to exploit reliability degradation. Task migration occurs when the kernel tries to balance workloads across CPUs. It forces a process to run on a different CPU, which effectively causes it to operate on another freelist because of the per-CPU freelist feature described in Section 2.1.

This inconsistency in the use of freelists can fail exploits. For example, a UAF exploit process allocates the vulnerable object on CPU-A and then gets migrated to CPU-B. When it frees the vulnerable object, the slot goes back to CPU-A's freelist. However, its payload object will be allocated from CPU-B's freelist, leading to unsuccessful exploitation. If the slot is acquired by a process running on CPU-A, the unexpected object overwrite can lead to kernel panic.

In short, unwanted task migration creates inconsistency in the use of per-CPU freelists, leading to exploit unreliability.

Unexpected Heap Usage. Modern multitasking kernels use context switches to simulate concurrency. When a task is switched out of the CPU, it waits for the next time slice.

In kernel heap exploits, some heap operations should be done atomically to achieve successful exploitation. A typical example is freeing the vulnerable object and reoccupying the freed slot in a UAF exploit. However, suppose a context switch happens after the object is freed and the next task occupies the target slot before the exploit process gets scheduled again. In that case, the exploit can never reacquire the target slot, leading to exploit failure.

In other words, unexpected heap usage may destroy an expected heap layout during runtime and introduce unreliability.

Unpredictable Corruption Timing. In the Linux kernel, many non-essential operations are delayed for the sake of performance. These operations will be scheduled by different components of the Linux kernel: softirq, workqueue, RCU, etc. Without access to the internal runtime information of these components, an exploit process cannot precisely predict when a specific operation will take place.

When a vulnerability involves these delayed operations, and more specifically, heap corruption happens inside a delayed operation, there is a time delay between vulnerability triggering and the heap operation taking effect. This poses difficulty on when to perform exploitation after the vulnerability is triggered. The common solution is to sleep for some time, making sure the delayed heap operation takes effect. However, this leaves more time for unexpected heap usage to happen and introduces unreliability.

6 Kernel Heap Exploit Model

Based on the discovered unreliability factors and the kernel attacks, we abstract the kernel heap exploit process as a *Kernel Heap Exploit Model*. This model explains the process of kernel heap exploitation, spanning from the moment that an exploit starts to interact with a vulnerable system to the moment that the exploit successfully triggers an attacker-controlled payload. It also includes all the possible failures occurring in the middle of the process. This model helps further zoom in on the cause of failure of a kernel heap exploit. In addition, it helps further delve into the reasons why a kernel heap stabilization technique succeeds or fails to improve kernel heap exploit reliability. Last but not least, it also helps instruct the development of new stabilization techniques or technique combinations that mitigate the unreliability issues for different types of kernel heap exploits.

As shown in Figure 1, the Kernel Heap Exploit Model is composed of 4 following stages:

1. **Context Setup.** An exploit prepares the necessary context to trigger a targeted vulnerability, *e.g.*, allocating a vulnerable object. Some kernel heap exploit stabilization techniques can be used in this stage to mitigate the unreliability factors (*e.g.*, unknown heap layout) and thus increase the exploit reliability.
2. **Vulnerability Effect Delay.** An exploit triggers the vulnerability by the end of context setup, but the heap layout may not change immediately. For example, for UAF and DF exploits, after an object free operation is invoked, the Linux kernel may delay the operation due to read-copy update (RCU) [15]. We call the period from when a vulnerability is triggered to when the corresponding heap operation takes effect *vulnerability effect delay*. Specifically, at this stage, the vulnerable object will be freed for UAF and DF exploits, the vulnerable object will be overflowed for OOB-Object exploits, and the freelist will be overwritten for OOB-Freelist exploits.
3. **Allocator Bracing.** After the heap operation takes effect, the memory allocator may transit to a corrupted state, especially for UAF, DF, and OOB-Freelist exploits. In this case, the exploit must restore the allocator from the corrupted state before the kernel detects the anomaly and turn panic, thus failing the exploit.
4. **Final Preparation.** At this stage, the unstable heap manipulation is finished. Some exploits will perform final preparations (*e.g.*, modifying payload objects) to obtain control flow hijacking primitives.

There are two following types of critical phrases that may contain unreliability factors:

1. **Slot-critical Phase.** This phase starts when a target slot is open and ends when the slot is filled by the target object. In this phase, the target slot is at risk of being occupied by other objects from other tasks.

   The slot-critical phase takes place at different stages for different types of exploits. For UAF and DF exploits, this phase happens after the vulnerability-triggering operations take effect, as the vulnerable object will be freed and other tasks may compete for the slot. For OOB-Object exploits and OOB-Freelist exploits, this phase happens before the vulnerability-triggering operations take effect, because OOB exploits allocate vulnerable objects before the vulnerability-triggering operations take effect.
2. **Allocator-Critical Phase.** In this phase, the allocator is in a corrupted state due to the operations of an exploit. This is critical to exploitation success because the kernel may detect the anomaly and panic when it tries to use the allocator. A successful exploit should restore the allocator from a corrupted state before it is caught by the kernel.

   Similar to the slot-critical phase, this phase also occurs at different stages for different types of exploits. For DF exploits, it happens at the same stage of the slot-critical phase because releasing a vulnerable object twice will corrupt the memory allocator. For UAF exploits, the allocator-critical phase occurs after the object release operation takes effect if the exploits tamper with the freelist. For OOB-Freelist, this phase happens after the slot-critical phase because overwriting the freelist will cause the allocator corruption issue. For OOB-Object, the allocator-critical phase does not exist because overflowing to the adjacent object does not invoke the memory allocator.

6.1 Exploit Stabilization Success and Failure

Using the Kernel Heap Exploit Model, we revisit the experiment results and investigate the success or failure of each technique. We discover that each technique improves exploitation reliability by mitigating at least one unreliability factor (Section 5.4), which suggests the comprehensiveness of the factor synthesis. In this subsection, we present the investigation result for each stabilization technique.

Defragmentation. This technique is used at the context setup stage when the kernel heap layout is unknown to the exploit. It drains the current per-CPU freelist and fills all half-full slabs to force the creation of a fresh freelist with a deterministic layout. Since OOB exploits require victim objects adjacent to the vulnerable objects, a deterministic freelist layout significantly increases the possibility of placing both objects at the right spots and improves exploit reliability.

UAF and DF exploits, on the other hand, do not require a specific heap layout. Therefore, the exploit reliability remains or even reduces in some cases, as illustrated in Table 3. We discover that this downgrade usually occurs when the technique is used *after* the allocation of vulnerable objects. After a vulnerable object is allocated, Defragmentation drains the current freelist and replaces the current freelist with a new freelist. In UAF and DF exploitation, when the vulnerable object is freed, the slot goes back to the slab freelist rather than

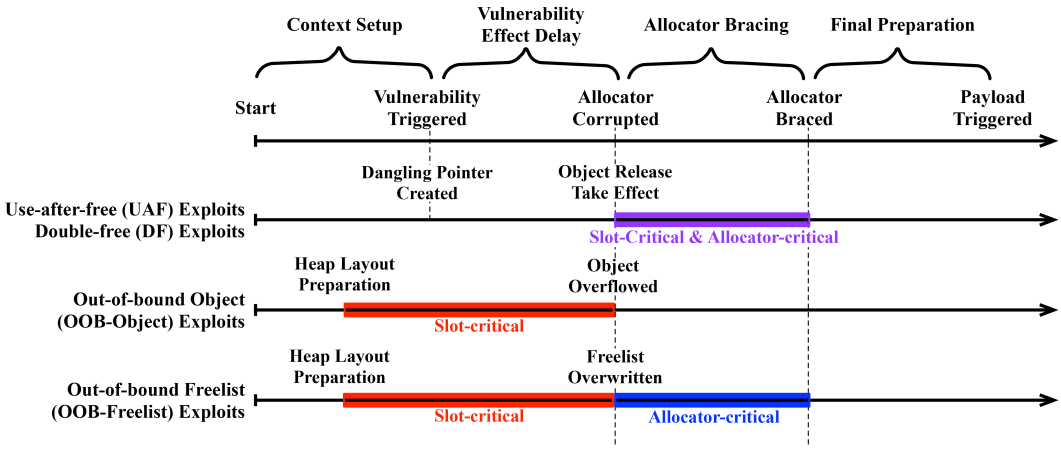

Figure 1: The Kernel Heap Exploit Model with different types of exploits and the critical phases.

the current per-CPU freelist, which poses difficulties in obtaining and overwriting the target slot, and thus downgrading exploit reliability.

Heap Grooming. Similar to Defragmentation, Heap Grooming works at the context setup stage and mitigates the Unknown Heap Layout unreliability factor. Consequently, Heap Grooming improves the reliability of OOB exploits.

Single-Thread Heap Spray. This technique improves exploit reliability by mitigating multiple unreliability factors mentioned above. These factors include Unknown Heap Layout, Unexpected Heap Usage, and Unpredictable Corruption Timing. To be more specific, Single-Thread Heap Spray mitigates Unknown Heap Layout by spraying a large number of payload objects, exhaustively searching for the target slot in the heap. It also remediates Unexpected Heap Usage since it can drain unexpected free slots and continue searching for the target slot during runtime. Furthermore, when the vulnerability involves delayed operations, the technique can also constantly allocate payload objects. This increases the chance to land a payload object when critical phases start, making an exploit resistant to Unpredictable Corruption Timing. Because of so many unreliability factors mitigated, as we observe from Table 3, Single-Thread Heap Spray is able to improve exploit reliability significantly in most cases.

Although Single-Thread Heap Spray demonstrates substantial reliability improvement, it can still fail if unexpected allocation or unwanted task migration happens during spray. For example, if an unexpected object allocation occupies the target slot, heap spray cannot reclaim the slot. Also, if the spray process is migrated to another CPU during runtime because of the per-CPU freelist feature (§2.1), it will inevitably spray on the new CPU's freelist, thus failing the exploit.

Multi-Process Heap Spray. Multi-Process Heap Spray is designed to mitigate the weakness of Single-Thread Heap Spray while preserving its strength. It occupies all CPU's runqueue (§2.1) with processes doing heap spray.

It mitigates unexpected allocations by preventing other processes, which may introduce unexpected allocations, from getting scheduled. This is because all CPU's runqueue is filled with spray processes. When the scheduler decides the next task to run, the task is likely a spray process.

Likewise, Multi-Process Heap Spray remediates Unwanted Task Migration by forcing the scheduler to run a spray process after the current spray process is migrated to another CPU.

Although this technique seems to mitigate all unreliability factors mentioned above, it works at a cost. Launching many processes shortens the time slice each process can have at a time, which increases the chance of occurrence for a context switch, resulting in more unexpected heap usage from the kernel itself. As a result, Multi-Process Heap Spray is a double-edged sword for exploit reliability. When the baseline exploit is unreliable, this technique is able to boost the reliability by a huge margin. However, when the baseline exploits are already reliable, it may introduce a slight reliability degradation to the exploits.

CPU Pinning. Due to the per-slab freelist implementation (§2.1), a freed slot goes back to its per-slab freelist. Exploit processes can migrate to different CPUs during execution due to task migration. When the exploit process frees a victim object, the target slot goes back to the original CPU's freelist, whereas its allocation happens in the new CPU's freelist. For this reason, the exploit process can never obtain the target slot, thus failing the exploitation.

CPU Pinning forces an exploit process to run on a specific CPU, which effectively forces it to use the same CPU freelist. Consequently, it is able to improve exploit reliability significantly in some cases.

7 New Technique and Compositions

As shown in Section 6.1, Kernel Heap Exploit Model helps us better understand the reason behind the success and failure of stabilization techniques. Going beyond enriching our under-

Type	CVE	Idle		Busy	
		Baseline	C.C.	Baseline	C.C.
OOB	2017-7184	33.51%	34.26% [+]	5.37%	10.11% [+]
OOB	2016-6187	32.89%	32.91% [+]	2.49%	16.03% [+]
OOB	2010-2959	22.09%	22.29% [+]	5.83%	39.06% [+]
OOB	2017-7308	0.04%	0.06% [+]	0.00%	0.00% [=]
UAF	2018-6555	89.98%	93.92% [+]	61.92%	82.08% [+]
UAF	2017-8824	97.64%	99.44% [+]	93.70%	98.82% [+]
UAF	2017-11176	92.17%	99.71% [+]	2.06%	11.51% [+]
DF	2017-2636	95.96%	96.06% [+]	93.26%	97.26% [+]
DF	2017-6074	98.50%	99.24% [+]	66.16%	93.68% [+]

Table 4: Exploit reliability of Context Conservation (C.C.)

Type	CVE	Baseline	Defragment	C.C.	D.+C.C.
In Idle State					
OOB	2017-7184	33.51%	99.46%	34.26%	99.23%
OOB	2016-6187	32.89%	99.63%	32.91%	99.63%
OOB	2010-2959	22.09%	99.37%	22.29%	99.46%
OOB	2017-7308	0.04%	0.32%	0.06%	13.14%
In Busy State					
OOB	2017-7184	5.37%	25.00%	10.11%	65.49%
OOB	2016-6187	2.49%	9.89%	16.03%	56.94%
OOB	2010-2959	5.83%	20.09%	39.06%	76.77%
OOB	2017-7308	0.00%	0.00%	0.00%	0.10%

Table 5: Exploit reliability of Defragmentation (Defragment), Context Conservation (C.C.), and its combination (D.+C.C.)

standing, the model can also help us create new techniques — in a solo or a compositing way — to stabilize exploits further. In this section, we present a new stabilization technique called Context Conservation and a series of combinations of stabilization techniques that improve exploit reliability.

7.1 Context Conservation

Taking advantage of Kernel Heap Exploit Model, we realize that context switches may influence the two critical phases in the way that the critical phases will last until the exploit process gets rescheduled and completed. This prolonged critical phase gives competitor processes more chance to occupy the target slot or crash the allocator.

Guided by insights derived from the model, we designed a novel stabilization technique — Context Conservation — that aims to avoid context switches during critical phases.

Context Conservation consists of two parts. First, it removes or relocates unnecessary code from critical phases. It is common to see code for context setup, debugging, and even sleep calls in critical phases. These unnecessary code snippets prolong the critical phase, increase the chance of context switch, and thus harm exploit reliability. By deleting or relocating the unnecessary code, one could shorten the critical phase and thus reduce the possibility that the context switch occurs in the critical phase.

Second, it injects a stub into the exploit process to predict when a fresh time slice could be obtained. With this information, one will dedicate the fresh slice to the critical phase and thus avoid a critical stage spanning across multiple time slices. This design could further decrease the possibility of context switch occurrence in the critical phase. Specifically, we design Context Conservation as a loop that runs a stub. In each iteration, the stub measures the CPU's time stamp counter (TSC). If the context switch does not occur within the loop, then each iteration takes fewer cycles. Otherwise, in specific iterations, the stub could observe many cycles caused by the CPU executing other processes. Using this cycle difference as an indicator, the security researchers could potentially dedicate a fresh time slice to the critical phase, ensuring no context switch takes a negative impact upon exploitation reliability.

To validate the novelty of Context Conservation, we checked with the interviewed experts, and we got the confirmation that this technique is novel. We also explored existing writeups and publicly available exploits and found no literature demonstrating this technique.

Applicability. Applying Context Conservation to kernel exploits requires the awareness of when the memory corruption occurs because the technique is designed to work during a critical phase. Therefore, Context Conservation cannot be applied to all vulnerabilities. Instead, it only works for exploits where the memory corruption time is predictable.

Evaluation. We evaluated Context Conservation on all applicable vulnerabilities and compared it against the baseline. As shown in Table 4, Context Conservation outperforms the baseline for all but one vulnerabilities and performs equally for the single exception.

Furthermore, we observe that Context Conservation does not improve the reliability of OOB exploits as effectively as UAF and DF. Heap layout is critical to OOB exploits as the victim object must be placed adjacent to the vulnerable object. While Context Conservation decreases heap layout changes caused by context switches, heap layout can still be changed by other factors. Therefore, the reliability of OOB exploits cannot be significantly improved by Context Conservation.

Inspired by the above investigation, we hypothesize that combining Context Conservation and Defragmentation will significantly improve OOB exploits as Defragmentation will help mitigate the heap layout issue caused by other factors. We validate this hypothesis and show the result in Table 5. As we can observe, the combination of Context Conservation and Defragmentation demonstrates a more significant improvement in exploitation reliability.

7.2 Compositing Stabilization Techniques

The kernel heap exploit model and the evaluation result indicate that each individual technique mitigates different unreliability factors to different degrees. As such, we further

Type	CVE	Idle			Busy			R.W. Tech	Combo Tech
		Baseline	Real-World	Combo	Baseline	Real-World	Combo		
OOB	2017-7533	20.43%	N/A	95.37% [+]	7.86%	N/A	82.60% [+]	N/A	①④⑤⑥
OOB	2017-7184	33.51%	N/A	97.86% [+]	5.37%	N/A	77.74% [+]	N/A	①④⑤⑥
OOB	2016-6187	32.89%	99.34% [+]	99.89% [+]	2.49%	0.46% [−]	60.63% [+]	①	①⑤⑥
OOB	2010-2959	22.09%	92.94% [+]	99.57% [+]	5.83%	88.54% [+]	95.09% [+]	①③	①④⑤⑥
OOB	2017-7308	0.04%	67.92% [+]	84.90% [+]	0.00%	0.00% [=]	0.76% [+]	①③	①④⑤⑥
UAF	2018-6555	89.98%	55.00% [−]	100.00% [+]	61.92%	58.18% [−]	100.00%[+]	③	④⑤⑥
UAF	2016-8655	96.06%	84.54% [−]	99.96% [+]	0.28%	83.28% [+]	99.54% [+]	③	④⑤
UAF	2017-15649	13.82%	99.68% [+]	99.96% [+]	18.34%	98.78% [+]	99.74% [+]	④	④⑤
UAF	2016-4557	99.34%	N/A	100.00% [+]	22.80%	N/A	99.09% [+]	N/A	④⑤
UAF	2017-8824	97.64%	65.16% [−]	99.56% [+]	93.70%	61.32% [−]	99.68% [+]	①④	④⑤⑥
UAF	2016-0728	1.48%	76.68% [+]	100.00% [+]	0.14%	34.22% [+]	99.48% [+]	④	④⑤
UAF	2017-10661	31.98%	44.96% [+]	95.96% [+]	0.18%	4.60% [+]	88.18% [+]	③	④⑤
UAF	2016-10150	0.74%	72.56% [+]	99.88% [+]	47.62%	81.20% [+]	88.48% [+]	④	④⑤
UAF	2017-11176	92.17%	98.23% [+]	100.00% [+]	2.06%	51.31% [+]	96.57% [+]	④⑤	④⑤⑥
DF	2017-2636	95.96%	N/A	99.80% [+]	93.26%	N/A	98.80% [+]	N/A	④⑤⑥
DF	2017-6074	98.50%	99.14% [+]	92.14% [−]	66.16%	89.76% [+]	87.66% [+]	③⑤	④⑤⑥
DF	2017-8890	51.96%	N/A	80.92% [+]	6.14%	N/A	72.04% [+]	N/A	④⑤

Table 6: Performance of the combined stabilization techniques in idle and busy settings. The numbers represent the following respect techniques: ① Defragmentation, ② Heap Grooming, ③ Single-Thread Heap Spray, ④ Multi-Process Heap Spray, ⑤ CPU Pinning, and ⑥ Context Conservation. The columns "R.W. Tech" and "Combo Tech" indicate the stabilization techniques combined in the real-world exploits and composite exploits, respectively. Note that for some vulnerabilities, the real-world exploits are unavailable. The reason is some real-world exploits are not end-to-end exploits. Although we derive baseline exploits from them, these incomplete real-world exploits, without major re-engineering, cannot perform successful end-to-end exploitation. We exclude these incomplete exploits from the evaluation.

explore the combination of multiple stabilization techniques and quantify the performance of the combined approach.

Based on our experiment result and Kernel Heap Exploit Model, we choose Defragmentation, CPU Pinning, Context Conservation, and Multi-Process Heap Spray to construct our stabilization composition technique. The rationale behind our selection is that, these 4 techniques nicely complement each other in mitigating different unreliability factors. Defragmentation eases the Unknown Heap Layout factor. CPU Pinning mitigates the impact of Unwanted Task Migration factor. Context Conservation and Multi-Process Heap Spray reduce the impact of Unexpected Heap Usage and Unknown Corruption Timing, respectively. It should be noted that we exclude Single-Thread Heap Spray and Heap Grooming because our aforementioned experiment result indicates that, overall, they underperform their alternative stabilization techniques — Defragmentation and Multi-Process Heap Spray (although they outperform other methods for some test cases).

Table 6 shows the performance of our combined stabilization technique. The combination of the 4 selected techniques cannot work for all vulnerabilities because the condition of triggering some vulnerabilities hinders the adoption of some techniques. For such scenarios, we exclude the corresponding stabilization method and preserve only those applicable.

Among all 17 vulnerabilities, the combined approach improves reliability for all baseline exploits in idle and busy settings except for the baseline exploit against the vulnerability CVE-2017-6074. Compared with the baseline exploits' success rates, the combined approach demonstrates a 135.53% exploitation reliability improvement (baseline: 38.61% vs. composition method: 90.94%). It indicates that the combined approach could be an effective method to stabilize exploits against nearly arbitrary vulnerabilities.

We also observe from Table 6 that exploits enhanced by our composition method also outperform the real-world exploits in terms of the exploitation success rate (real-world: 54.30% vs. composition method: 91.15%). In this work, we gathered these real-world exploits from the Internet. These exploits are crafted by professional security researchers. As such, the superiority over real-world exploits implies that the stabilization method derived from a systematic study could better improve an exploit's stabilization than ad-hoc methods commonly adopted by security practitioners.

We investigated the exploit for CVE-2016-10150 to understand why the combined technique improves its reliability drastically, especially in busy systems. In this exploit, the exploit thread and the spray thread must remain different due to the nature of the vulnerability. Since the exploit thread and the spray thread both do intensive work, the Linux kernel will mark them as both heavy tasks and migrate one of them to another CPU for workload balancing, which hurts exploit reliability. Our investigation shows that this is also the reason why Single-Thread Heap Spray performs worse than the baseline for this CVE: its spray thread performs more intensive work and is more likely to be migrated compared with the baseline. CPU Pinning forces the two threads to run

on the same CPU, thus avoiding the unwanted migration and improving reliability significantly as shown in Table 3. With both threads pinned to the same CPU, adding Multi-Process Heap Spray increases the chance of hitting the target slot and further improves the reliability.

Our composition method fails to improve reliability for the baseline exploit against CVE-2017-6074, especially in the idle setting. In addition, for the same case, the composition-method-enhanced exploit outperforms baseline exploits in the busy setting. However, its reliability improvement is lower than that of the real-world exploit. To understand the reason behind this observation, we manually inspect both exploits. Our discovery aligns with our early conclusion — Multi-Process Heap Spray is a double-edge sword. In the idle setup, it introduces unnecessary additional workload, harming the reliability of the baseline exploit. On the contrary, in the busy mode, though Multi-Process Heap Spray imposes additional workload, which harms exploitation reliability, it also reduces the impact of workload imposed by other processes, improving the baseline exploit reliability to some extent. This explains why the real-world exploit using Single-Thread Heap Spray demonstrates better reliability improvement.

8 Discussion and Future Work

Insights on Defense. Due to limited manpower, patches to vulnerabilities are prioritized by the security impact of the reported vulnerabilities [41]. In current security impact assessment systems (*i.e.*, Common Vulnerability Scoring System [14], or CVSS score in short), exploit reliability serves as an important factor. Misunderstanding exploit reliability may mislead developers by presenting a severe vulnerability that can be exploited reliably as low security impact. In practice, the false security impact assessment may mislead downstream kernel developers and lead them to de-prioritize the patches to severe vulnerabilities because of the low CVSS score. Our systematic study assists developers to clearly understand the kernel exploit reliability issue, helping them to correctly prioritize patches to vulnerabilities.

Real-world Systems. We used two system workloads, namely, idle and busy, to demonstrate reliability variation under different workloads for each stabilization technique. Such a setup does not reflect the reliability of each stabilization technique on real-world systems because real-world systems can have idle and busy workloads intertwining with each other. As part of our future work, we will design more systematic experiments to understand exploit reliability under the real-world workload.

Kernel Heap Exploit Reliability for Other Operating Systems. This paper focuses on Linux operating systems, aiming to understand the kernel heap exploit reliability issues rooted in Linux kernel. However, kernel heap exploit reliability issue also exists in other operating systems such as Windows. For example, Windows exploit developers propose to use heap data repairing and memory alignment to improve the reliability of exploit against CVE-2015-0057 in Windows [40]. We will study this research problem for other operating systems in the future.

Exploit Stabilization Techniques Combinations. Another research direction is to study the composition of exploit stabilization techniques to further improve exploit reliability. As pointed out in Section 7.2, combining multiple individual exploit stabilization techniques helps to improve exploit reliability. However, it is hard to justify that our method is the *best* combination because one should evaluate all the possible technique combinations to draw the conclusion.

One may argue that instead of trying all combinations, we could add techniques incrementally. For example, we could test Defragmentation with Heap Grooming first; if the result is better than Defragmentation alone, we continue to add more techniques; otherwise, we withdraw Heap Grooming. However, such an incremental approach will not work. In our study, we observe that techniques are not independent of each other. Some techniques rely on the existence of other techniques to further improve reliability; themselves alone do not improve reliability significantly in some settings. Moreover, techniques behave differently for different vulnerability types. There may not be a single combination that performs the best for all vulnerabilities. Given the problem complexity, we leave the study of combinations of exploit stabilization techniques as separate research for future work.

9 Related Work

Here, we summarize the works most relevant to ours and discuss their difference from ours.

Kernel Exploitability Assessment. There are a rich collection of research works on kernel exploitation. From technical perspectives, kernel exploitability assessment research could be categorized into three sub-areas below.

The first research area is to explore the capability of vulnerabilities. FUZE [43] searches for new use sites of UAF vulnerabilities using under-context fuzzing and further employs symbolic execution to identify exploitable primitives implied by the new use sites. KOOBE [20] retrieves the corruption capabilities of an OOB vulnerability manifested in the PoC programs and discloses hidden capabilities using capability-guided fuzzing. Yonchan et al. [32] increase the success rate of triggering race condition vulnerabilities by invoking inter-process interrupts to enlarge the time window.

The second research area is to obtain exploitable primitives. For UAF vulnerabilities, Xu et al. [44] use two memory collision mechanisms to occupy the freed memory region in the kernel. Lu et al. [34] exploit use-before-initialization vulnerabilities using deterministic stack spraying and reliable exhaustive memory spraying. To take one step forward, Cho et al. [23] further propose to make use of eBPF functionality in the kernel for stack spraying. SLAKE [22] assists the

exploitation of slab-based vulnerabilities by first building a database of kernel objects and then systematically manipulating the layout of kernel heap by using the kernel objects in the database.

The third research area is to bypass mitigations in the kernel. ret2dir [30] utilizes physical memory mapped to kernel space for payload injection. KEPLER [42] uses communication channels between kernel space and user space to leak stack canary and inject payload to kernel stack for ROP programming. ELOISE [21] leverages a special but prevailing type of structure to bypass KASLR and heap cookie protector. Relying on hardware side channel attacks, [25, 28, 29, 33] circumvent KASLR without triggering SMEP/SMAP.

Exploit Generation Techniques. In the past decade, researchers have explored exploit generation approaches.

Brumley et al. [17, 19] proposed preconditioned symbolic execution to automatically generate exploits for stack overflow and format string vulnerabilities. Bao et al. [18] developed shellcode layout remediation and path kneading approaches that automatically transplant existing shellcode to the target vulnerability. The Shellphish team designed PovFuzzer and Rex to turn crashes into exploits [36, 37, 38]. PovFuzzer keeps mutating input and, at the same time, observing how the input influences the crash. Following this work, Rex symbolically executes the input with the purpose of jumping to a shellcode prepared in advance. Heelan et al. focus on heap buffer overflow vulnerabilities in user space programs. SHRIKE [26] uses regression tests to learn how to automate heap layout manipulation and corrupt the sensitive pointers. They further improve their proposed approach by using a genetic algorithm to replace the random search algorithm in Gollum [27] for exploiting heap overflow vulnerabilities in language interpreters. With a similar goal, Revery [39] also explores exploitable memory layouts for vulnerabilities in user space programs by using fuzz testing along with a program synthesis method to guide the construction of a working exploit. HeapHopper [24] and Insu et al. [45] discover new exploitation primitives in the heap allocator, providing heap operations and attack capabilities as actions, driving the heap allocator to execute until primitives such as arbitrary write or overlapped chunks are identified.

Our work is fundamentally different from those above. Rather than concentrating on exploit generation, our work studies the reliability problem in exploitation. To the best of our knowledge, this is the first work that explores and studies exploitation stabilization techniques.

10 Conclusion

The reliability of Linux kernel exploits heavily rely on security researchers' experiences and their opinions on exploit stabilization techniques. In this research, we surprisingly discover that security researchers' opinions on exploit stabilization techniques are diverse. Besides, we also observe that even those security researchers with extensive experience in Linux kernel exploitation research have incorrect intuitions towards how to use exploit stabilization techniques and under what conditions the stabilization techniques could work. Through a comprehensive, systematic study, this research work analyzes existing stabilization techniques thoroughly and identifies the working condition of each exploit stabilization technique. We show that the discovery from our study could help experienced security researchers better understand stabilization techniques. In addition to the study, this research also proposes a new technique to improve exploit reliability. We show that by combining the new technique with existing stabilization techniques, we can craft exploits demonstrating much higher reliability than those crafted by security researchers. With this discovery in hand, we safely conclude that adopting exploit stabilization techniques correctly could significantly improve an exploit's reliability and thus benefit the accuracy of exploitability assessment.

11 Acknowledgement

We would like to thank our shepherd, Vasileios Kemerlis, and the anonymous reviewers for their helpful feedback. This material was supported by grants from Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0060, HR00112190093, and FA875019C0003, the Army Research Office (ARO) under Grant No. W911NF-17-1-0370, the National Science Foundation (NSF) under Grant No. 1954466 and 2000792, the Office of Naval Research (ONR) under Grant No. N00014-20-1-2008, the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under Grant No. NRF-2021R1A4A1029650, and the 2020 IBM PhD Fellowship Program.

References

[1] CFS Scheduler — The Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[2] cve-2016-6187-poc/matreshka.c at master. https://github.com/vnik5287/cve-2016-6187-poc/blob/master/matreshka.c.

[3] cve-2017-11176/cve-2017-11176.c at master. https://github.com/lexfo/cve-2017-11176/blob/master/cve-2017-11176.c.

[4] kernel-exploitation/cve-2021-32606.md at main. https://github.com/nrb547/kernel-exploitation/blob/main/cve-2021-32606/cve-2021-32606.md.

[5] Linux Kernel 4.4.0 (Ubuntu 14.04/16.04 x86-64) - AF_PACKET Race Condition Privilege Escalation - Linux_x86-64 local Exploit. https://www.exploit-db.com/exploits/40871.

[6] Linux Kernel 4.9 - TCP Socket Handling Use-After-Free (CVE-2019-15239). https://pulsesecurity.co.nz/advisories/linux-kernel-4.9-tcpsocketsuaf.

[7] linux-kernel-exploits/cve-2016-0728.c at master. https://github.com/SecWiki/linux-kernel-exploits/blob/master/2016/CVE-2016-0728/cve-2016-0728.c.

[8] Linux_kernel_exploits/exp.c at master. https://github.com/ww9210/Linux_kernel_exploits/blob/master/cve-2016-10150/exp.c.

[9] Linux_kernel_exploits/exploit.c at master. https://github.com/ww9210/Linux_kernel_exploits/blob/master/cve-2017-15649/exploit.c.

[10] New Old Bugs in the Linux Kernel. https://blog.grimm-co.com/2021/03/new-old-bugs-in-linux-kernel.html.

[11] offensive_poc/exploit.c at master. https://github.com/hardenedlinux/offensive_poc/blob/master/CVE-2017-7533/exploit.c.

[12] p4nda's blog. http://p4nda.top/2019/02/16/CVE-2017-7184/.

[13] SSD Advisory – IRDA Linux Driver UAF - SSD Secure Disclosure. https://ssd-disclosure.com/ssd-advisory-irda-linux-driver-uaf/.

[14] What are CVSS Scores | Balbix. https://www.balbix.com/insights/understanding-cvss-scores/.

[15] What is RCU, Fundamentally? https://lwn.net/Articles/262464/.

[16] Phoronix test suite, 2015. http://www.phoronix-test-suite.com/.

[17] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2011.

[18] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[19] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[20] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[21] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[22] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[23] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting uses of uninitialized stack variables in Linux kernels to leak kernel pointers. In *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[24] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. {HeapHopper}: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, 2018.

[25] Daniel Gruss, Clémentine Maurice, and Anders Fogh. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[26] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[27] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[30] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[31] Andrey Konovalov. Linux kernel exploitation, 2020. https://github.com/xairy/linux-kernel-exploitation.

[32] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. Exploiting kernel races through taming thread interleaving. In *BlackHat USA*, 2020.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[34] Kangjie Lu, Marie-Therese Walter, David Pfaff, and Stefan Nürnberger and Wenke Lee and Michael Backes. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[35] Gene Novark and Emery D Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 573–584, Chicago, IL, October 2010.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[38] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[39] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, BingChang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[40] Yu Wang. A new CVE-2015-0057 exploit technology. In *BlackHat Asia*, 2015.

[41] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.

[42] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[43] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Wei Zou, and Xiaorui Gong. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[44] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in Linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[45] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

Appendix-A: Detail of Experiment Setup

Here, we present more details about our experiment setup.

Dataset. To obtain data, namely vulnerabilities and exploits, we exhaustively search for them online for our experiment. We collect public exploits from different sources, including exploit writeups from organizations [3, 7, 13], individuals [2, 5, 12, 31], and released repositories of previous research works [21, 22, 42, 43] focusing on Linux kernel exploitation. Next, we screen the vulnerabilities and exploits with the following criteria for the dataset: ❶ The public exploits demonstrate the potential of obtaining control flow hijacking primitive, which indicates successful exploitation. ❷ The vulnerabilities can be backported to v4.15 kernel, and their corresponding exploits still preserve the potential of obtaining control flow hijacking primitive on the new kernel after minor fixes. This rules out reliability differences introduced by different kernel versions. ❸ The vulnerabilities do not rely on special hardware devices or emulators to trigger for the ease of experiment. Following the selection criteria above and finish incomplete exploits by ourselves, we form a dataset containing 17 Linux kernel vulnerabilities and their corresponding public exploits.

Exploit variants. Based on the public exploits, we construct exploits used for our experiment by following the steps below. First, we build baseline exploits by stripping away stabilization techniques already implemented in the public exploits. In this process, we do not find new stabilization techniques in public exploits that is not reported by the interview participants. Second, we craft exploit variants by adding individual stabilization technique into the corresponding baseline exploit. In this process, we follow the implementation suggestions gathered from the participants. It ensures that each exploit variant encloses only one stabilization technique in the way where security researchers usually follow.

During exploit variants construction, we keep the modifications on baseline exploits to the minimum in order to avoid unexpected impacts on reliability caused by exploit structure change. As is mentioned in Section 5.1, we eventually obtain 102 exploits for our evaluation. Note that we skip the construction of exploit variants if the technique cannot be implemented in the corresponding vulnerability. For example, heap spray cannot be applied to CVE-2016-6187 because the exploit aims to corrupt adjacent freelist metadata, and heap spray eliminates the attacking surface.

VM vs. bare-metal machine. As is mentioned in Section 5.1, we run each exploit on a VM. Admitted that the success rate of each exploit we obtain from VMs may be different from that on bare-metal machines, we argue that the change of reliability (*e.g.*, improvement or degradation) of stabilization techniques is consistent. Therefore, by observing the results from VMs, we can safely draw conclusions.

Appendix-B: More Experiment Results

Table 3 shows our experiment result obtained in the configuration of 2 CPUs and 2 GB RAM. As is mentioned in Section 5.1, we varied our configuration. Table 7, 8, and 9 illustrate our experiment results observed in other configurations.

1. Which of the following techniques have you adopted to stabilize your exploits – (A) Defragmentation, (B) Heap Grooming, (C) Single-Thread Heap Spray, (D) Multi-Process Heap Spray, (E) CPU Pinning?
2. If you've never heard about the stabilization techniques above, please report them to us.
3. if you've heard about the stabilization techniques above but have no clue how to use them to improve exploit reliability, please report them to us as well.
4. What other approaches have you ever used to make your Linux kernel exploit reliable?
5. Why do you think the techniques you've used could work for stabilizing exploits?
6. In what conditions, the stabilization techniques you are familiar with are effective? (*e.g.*, in lightweight or heavy workload)
7. Do the stabilization techniques familiar to you work only for specific types of vulnerabilities? If so, please specify which technique and for what type of vulnerabilities?
8. Comparing the defragmentation technique with heap grooming technique, which one is more potent in terms of improving exploit reliability?
9. How do you implement your exploitation stabilization techniques?
10. Among all the techniques you mentioned, which one do you think is the most effective?
11. Have you ever discovered that when you apply an exploitation stabilization technique for a vulnerability, the exploit reliability is not improved but jeopardized? Have you tried to debug it? What is the reason behind this discovery?

Figure 2: Open questions used in our interview.

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
In Idle State							
OOB	2017-7533	24.72%	52.42% [+]	N/A	99.36% [+]	99.62% [+]	36.42% [+]
OOB	2017-7184	30.72%	99.78% [+]	100.00% [+]	46.74% [+]	96.96% [+]	64.12% [+]
OOB	2016-6187	51.86%	99.88% [+]	N/A	N/A	N/A	67.94% [+]
OOB	2010-2959	15.48%	99.88% [+]	99.96% [+]	65.22% [+]	66.26% [+]	21.52% [+]
OOB	2017-7308	0.02%	0.00% [−]	N/A	0.00% [−]	4.64% [+]	0.04% [+]
UAF	2018-6555	96.52%	82.24% [−]	N/A	100.00% [+]	100.00%[+]	98.56% [+]
UAF	2016-8655	97.18%	96.08% [−]	N/A	99.74% [+]	99.80% [+]	99.56% [+]
UAF	2017-15649	17.00%	17.98% [+]	N/A	73.56% [+]	99.86% [+]	99.98% [+]
UAF	2016-4557	99.76%	97.94% [−]	N/A	100.00%[+]	99.98%[+]	99.80% [+]
UAF	2017-8824	97.96%	0.04% [−]	N/A	97.46% [−]	99.24% [+]	99.98% [+]
UAF	2016-0728	1.74%	2.32% [+]	N/A	82.20% [+]	100.00% [+]	2.68% [+]
UAF	2017-10661	18.80%	20.00% [+]	N/A	35.38% [+]	80.86% [+]	73.66% [+]
UAF	2016-10150	0.00%	0.00% [=]	N/A	0.00% [=]	99.64% [+]	99.98% [+]
UAF	2017-11176	96.08%	7.36% [−]	N/A	99.96% [+]	99.10% [+]	88.70% [−]
DF	2017-2636	99.42%	14.04% [−]	N/A	99.86% [+]	99.72% [+]	99.70% [+]
DF	2017-6074	99.98%	95.14% [−]	N/A	99.86% [−]	94.74% [−]	99.88% [−]
DF	2017-8890	56.44%	0.30% [−]	N/A	58.52% [+]	75.66% [+]	76.98% [+]
In Busy State							
OOB	2017-7533	5.56%	10.52% [+]	N/A	71.58% [+]	97.74% [+]	13.14% [+]
OOB	2017-7184	9.78%	51.42% [+]	64.98% [+]	44.96% [+]	89.32% [+]	9.48% [−]
OOB	2016-6187	5.36%	53.78% [+]	N/A	N/A	N/A	10.78% [+]
OOB	2010-2959	9.96%	47.54% [+]	32.84% [+]	36.04% [+]	65.72% [+]	19.06% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.12% [+]	0.00% [=]
UAF	2018-6555	72.44%	66.86% [−]	N/A	86.10% [+]	99.24% [+]	80.70% [+]
UAF	2016-8655	0.20%	0.22% [+]	N/A	33.56% [+]	99.76% [+]	4.64% [+]
UAF	2017-15649	10.32%	10.18% [−]	N/A	38.50% [+]	99.82% [+]	99.72% [+]
UAF	2016-4557	51.82%	59.92% [+]	N/A	97.48% [+]	99.84% [+]	60.14% [+]
UAF	2017-8824	85.66%	0.06% [−]	N/A	86.56% [+]	98.46% [+]	100.00% [+]
UAF	2016-0728	0.04%	0.02% [−]	N/A	24.06% [+]	99.98% [+]	0.02% [−]
UAF	2017-10661	0.04%	0.02% [−]	N/A	4.12% [+]	69.80% [+]	0.76% [+]
UAF	2016-10150	48.12%	47.90% [−]	N/A	43.70% [−]	49.30% [+]	97.88% [+]
UAF	2017-11176	1.90%	2.32% [+]	N/A	41.52% [+]	94.24% [+]	3.72% [+]
DF	2017-2636	96.46%	24.38% [−]	N/A	99.00% [+]	99.34% [+]	99.56% [+]
DF	2017-6074	74.00%	62.48% [−]	N/A	75.40% [+]	93.04% [+]	78.22% [+]
DF	2017-8890	19.98%	1.24% [−]	N/A	29.08% [+]	49.28% [+]	34.58% [+]

Table 7: Performance of stabilization techniques under the configuration of 4 CPUs and 2G memory.

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
In Idle State							
OOB	2017-7533	20.58%	48.50% [+]	N/A	98.40% [+]	99.60% [+]	31.42% [+]
OOB	2017-7184	32.74%	99.26% [+]	100.00% [+]	45.54% [+]	96.42% [+]	53.68% [+]
OOB	2016-6187	31.96%	99.46% [+]	N/A	N/A	N/A	31.54% [−]
OOB	2010-2959	22.16%	99.66% [+]	99.88% [+]	56.10% [+]	58.68% [+]	21.50% [−]
OOB	2017-7308	0.08%	0.30% [+]	N/A	0.00% [−]	4.68% [+]	0.10% [+]
UAF	2018-6555	89.82%	77.24% [−]	N/A	99.72% [+]	100.00%[+]	93.78% [+]
UAF	2016-8655	96.42%	94.10% [−]	N/A	99.44% [+]	99.80% [+]	99.18% [+]
UAF	2017-15649	13.72%	12.76% [−]	N/A	58.62% [+]	99.88% [+]	99.60% [+]
UAF	2016-4557	99.26%	98.24% [−]	N/A	100.00%[+]	99.98%[+]	99.36% [+]
UAF	2017-8824	97.76%	0.20% [−]	N/A	98.24% [+]	98.82% [+]	98.88% [+]
UAF	2016-0728	1.98%	6.60% [+]	N/A	77.58% [+]	100.00% [+]	3.44% [+]
UAF	2017-10661	33.80%	32.24% [−]	N/A	51.24% [+]	82.18% [+]	71.48% [+]
UAF	2016-10150	1.10%	0.86% [−]	N/A	1.56% [+]	99.72% [+]	99.48% [+]
UAF	2017-11176	92.66%	13.06% [−]	N/A	99.90% [+]	99.92% [+]	97.34% [+]
DF	2017-2636	96.14%	21.72% [−]	N/A	99.48% [+]	99.32% [+]	98.12% [+]
DF	2017-6074	98.76%	84.12% [−]	N/A	99.00% [+]	91.88% [−]	98.54% [−]
DF	2017-8890	51.58%	0.70% [−]	N/A	67.16% [+]	74.68% [+]	80.22% [+]
In Busy State							
OOB	2017-7533	7.72%	23.80% [+]	N/A	75.26% [+]	80.02% [+]	15.90% [+]
OOB	2017-7184	5.34%	27.56% [+]	53.88% [+]	35.26% [+]	72.72% [+]	7.50% [+]
OOB	2016-6187	2.30%	10.24% [+]	N/A	N/A	N/A	5.26% [+]
OOB	2010-2959	5.30%	18.46% [+]	46.44% [+]	19.02% [+]	43.90% [+]	9.68% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.20% [+]	0.00% [=]
UAF	2018-6555	60.48%	61.72% [+]	N/A	80.36% [+]	96.42% [+]	73.92% [+]
UAF	2016-8655	0.36%	0.42% [+]	N/A	58.54% [+]	99.74% [+]	0.42% [+]
UAF	2017-15649	17.94%	16.82% [−]	N/A	35.98% [+]	99.22% [+]	92.12% [+]
UAF	2016-4557	23.72%	14.06% [−]	N/A	94.56% [+]	99.06% [+]	19.74% [−]
UAF	2017-8824	93.90%	0.38% [−]	N/A	95.16% [+]	98.98% [+]	98.70% [+]
UAF	2016-0728	0.14%	0.18% [+]	N/A	32.84% [+]	99.74% [+]	0.08% [−]
UAF	2017-10661	0.24%	0.26% [+]	N/A	28.44% [+]	69.08% [+]	0.24% [=]
UAF	2016-10150	46.02%	46.94% [+]	N/A	39.28% [−]	62.26% [+]	77.04% [+]
UAF	2017-11176	1.70%	1.58% [−]	N/A	39.56% [+]	92.30% [+]	4.38% [+]
DF	2017-2636	92.98%	18.98% [−]	N/A	97.74% [+]	99.18% [+]	97.86% [+]
DF	2017-6074	65.46%	50.04% [−]	N/A	67.56% [+]	74.94% [+]	65.70% [+]
DF	2017-8890	6.16%	1.60% [−]	N/A	20.26% [+]	44.74% [+]	9.50% [+]

Table 8: Performance of stabilization techniques under the configuration of 2 CPUs and 4G memory.

Type	CVE	Baseline	Defragmentation	Heap Grooming	Single-Thread Heap Spray	Multi-Process Heap Spray	CPU Pinning
In Idle State							
OOB	2017-7533	22.60%	51.94% [+]	N/A	99.49% [+]	99.49% [+]	38.54% [+]
OOB	2017-7184	32.49%	99.86% [+]	100.00% [+]	44.11% [+]	97.49% [+]	64.97% [+]
OOB	2016-6187	54.94%	99.69% [+]	N/A	N/A	N/A	62.51% [+]
OOB	2010-2959	15.40%	99.89% [+]	99.94% [+]	64.63% [+]	64.80% [+]	20.71% [+]
OOB	2017-7308	0.03%	0.03% [=]	N/A	0.00% [−]	4.69% [+]	0.11% [+]
UAF	2018-6555	96.71%	82.89% [−]	N/A	99.97% [+]	100.00%[+]	98.51% [+]
UAF	2016-8655	97.26%	95.94% [−]	N/A	99.80% [+]	99.74% [+]	99.54% [+]
UAF	2017-15649	15.69%	16.66% [+]	N/A	73.31% [+]	99.89% [+]	100.00%[+]
UAF	2016-4557	99.71%	97.83% [−]	N/A	100.00%[+]	100.00%[+]	99.77% [+]
UAF	2017-8824	97.89%	0.03% [−]	N/A	97.77% [−]	99.26% [+]	100.00%[+]
UAF	2016-0728	1.71%	2.20% [+]	N/A	82.80% [+]	99.97% [+]	2.49% [+]
UAF	2017-10661	18.83%	18.31% [−]	N/A	35.20% [+]	76.43% [+]	72.89% [+]
UAF	2016-10150	0.00%	0.00% [=]	N/A	0.00% [=]	99.49% [+]	100.00%[+]
UAF	2017-11176	91.89%	8.34% [−]	N/A	99.97% [+]	99.17% [+]	96.46% [+]
DF	2017-2636	99.54%	12.80% [−]	N/A	99.83% [+]	99.66% [+]	99.86% [+]
DF	2017-6074	99.86%	94.63% [−]	N/A	99.94% [+]	94.31% [−]	99.94% [+]
DF	2017-8890	57.89%	0.26% [−]	N/A	61.46% [+]	76.37% [+]	76.31% [+]
In Busy State							
OOB	2017-7533	4.37%	10.34% [+]	N/A	73.40% [+]	97.37% [+]	10.80% [+]
OOB	2017-7184	9.86%	51.94% [+]	65.89% [+]	44.66% [+]	89.26% [+]	8.37% [+]
OOB	2016-6187	4.54%	24.34% [+]	N/A	N/A	N/A	12.37% [+]
OOB	2010-2959	10.69%	49.94% [+]	31.29% [+]	36.06% [+]	65.91% [+]	18.94% [+]
OOB	2017-7308	0.00%	0.00% [=]	N/A	0.00% [=]	0.09% [+]	0.00% [=]
UAF	2018-6555	73.74%	67.40% [+]	N/A	85.43% [+]	99.60% [+]	81.89% [+]
UAF	2016-8655	0.06%	0.14% [+]	N/A	32.06% [+]	99.86% [+]	4.03% [+]
UAF	2017-15649	9.66%	10.83% [−]	N/A	38.80% [+]	99.80% [+]	99.40% [+]
UAF	2016-4557	49.63%	45.11% [−]	N/A	97.20% [+]	99.77% [+]	59.31% [−]
UAF	2017-8824	86.91%	0.06% [−]	N/A	86.94% [+]	99.46% [+]	99.94% [+]
UAF	2016-0728	0.17%	0.00% [+]	N/A	24.23% [+]	100.00% [+]	0.06% [−]
UAF	2017-10661	0.03%	0.09% [+]	N/A	3.74% [+]	69.20% [+]	0.80% [=]
UAF	2016-10150	48.11%	47.17% [+]	N/A	45.83% [−]	48.40% [+]	97.20% [+]
UAF	2017-11176	1.71%	1.91% [−]	N/A	35.34% [+]	92.94% [+]	22.40% [+]
DF	2017-2636	93.69%	23.60% [−]	N/A	97.89% [+]	99.43% [+]	99.63% [+]
DF	2017-6074	74.49%	63.91% [−]	N/A	74.91% [+]	93.11% [+]	77.34% [+]
DF	2017-8890	19.57%	1.40% [−]	N/A	32.57% [+]	51.54% [+]	34.77% [+]

Table 9: Performance of stabilization techniques under the configuration of 4 CPUs and 4G memory.