Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability
Kyle Zeng, Arizona State University; Yueqi Chen, Pennsylvania State University; Haehyun Cho, Arizona State University and Soongsil University; Xinyu Xing, Pennsylvania State University; Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao, Arizona State University
https://www.usenix.org/conference/usenixsecurity22/presentation/zeng

This paper is included in the Proceedings of the 31st USENIX Security Symposium.
August 10–12, 2022 • Boston, MA, USA
978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.
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...
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.

## 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

\(^1\)https://github.com/sefcom/KHeaps
flow. For example, it can overwrite function pointers or hijack virtual table pointers adjacent to the overflown 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.

### 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
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 right next to a vulnerable object.

3. **Heap Spray** utilizes the technique of locating many victim objects and then swapping one victim object right next to a vulnerable object.

4. **Multi-Process Heap Spray** allows the attacker to exploit multiple processes for executing the technique code, etc.

5. **Defragmentation** initializes the kernel heap by allocating a large number of objects in the same cache as the vulnerable object, forcing the allocator to create new slabs.

6. **Heap Spray** locates many victim objects and then swaps one victim object right next to a vulnerable object, creating a heap layout beneficial for some heap exploits.

7. **Multi-Process Heap Spray** executes the technique code in multiple processes for better performance.

8. **CPU Pinning** ensures the threads/processes for exploit are pinned to idle CPUs, improving exploitation reliability.

Their effectiveness can be improved by setting appropriate 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.
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.

### 4.2 Interview Responses

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

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

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 Evaluating Stabilization Techniques

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-
With the platform, we can automatically run all exploits for the 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 (\texttt{kmalloc} and \texttt{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 detail of generating baseline exploits and exploits armed with each stabilization technique in Appendix-A. The numbers may not reflect their real success rates. These exploits are marked with \([\ast]\) means improvement over baseline, \([\ast\ast]\) is for degradation, and \([\ast\ast\ast]\) 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 \([\ast]\).

### 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 workload constantly occupying 81.24% CPU usage on all CPUs and imposing intensive pressure on the kernel heap. The background heap operations (\texttt{kmalloc} and \texttt{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 freeslit.

### 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 of, in other words, the factors attributable to exploit unreliability.

**Unknown Heap Layout.** Before the execution of an exploit, other processes may shuffle the freeslit 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 freeslit 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 freeslit because of the per-CPU freeslit 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 freeslit. However, its payload object will be allocated from CPU-B’s freeslit, leading to unsuccessful exploitation.

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 over-written 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 overwriting 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
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 underst-
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 pro-
cess 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 re-
moves 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 snip-
ets prolong the critical phase, increase the chance of context
switch, and thus harm exploit reliability. By deleting or relo-
cating 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 informa-
tion, 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 de-
sign 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 dedi-
cate a fresh time slice to the critical phase, ensuring no context
switch takes a negative impact upon exploitation reliability.

7.1 Context Conservation

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

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Baseline</th>
<th>C.C.</th>
<th>Baseline</th>
<th>C.C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOB</td>
<td>2017-7184</td>
<td>33.51%</td>
<td>99.63%</td>
<td>33.51%</td>
<td>99.23%</td>
</tr>
<tr>
<td>OOB</td>
<td>2016-6187</td>
<td>32.89%</td>
<td>99.63%</td>
<td>32.89%</td>
<td>99.23%</td>
</tr>
<tr>
<td>OOB</td>
<td>2017-2636</td>
<td>22.09%</td>
<td>99.63%</td>
<td>22.09%</td>
<td>99.63%</td>
</tr>
<tr>
<td>OOB</td>
<td>2017-7308</td>
<td>0.04%</td>
<td>0.06%</td>
<td>0.04%</td>
<td>0.06%</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-8824</td>
<td>97.64%</td>
<td>99.63%</td>
<td>97.64%</td>
<td>99.63%</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-11176</td>
<td>92.17%</td>
<td>99.63%</td>
<td>92.17%</td>
<td>99.63%</td>
</tr>
<tr>
<td>DF</td>
<td>2017-2636</td>
<td>95.96%</td>
<td>99.63%</td>
<td>95.96%</td>
<td>99.63%</td>
</tr>
<tr>
<td>DF</td>
<td>2017-7308</td>
<td>98.50%</td>
<td>99.63%</td>
<td>98.50%</td>
<td>99.63%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Baseline</th>
<th>Defragment</th>
<th>C.C.</th>
<th>D.+C.C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOB</td>
<td>2017-7184</td>
<td>5.37%</td>
<td>0.00%</td>
<td>99.11%</td>
<td>99.24%</td>
</tr>
<tr>
<td>OOB</td>
<td>2016-6187</td>
<td>2.49%</td>
<td>0.00%</td>
<td>99.11%</td>
<td>99.24%</td>
</tr>
<tr>
<td>OOB</td>
<td>2017-2636</td>
<td>2.49%</td>
<td>0.00%</td>
<td>99.11%</td>
<td>99.24%</td>
</tr>
<tr>
<td>OOB</td>
<td>2017-7308</td>
<td>0.04%</td>
<td>0.00%</td>
<td>99.11%</td>
<td>99.24%</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-8824</td>
<td>97.64%</td>
<td>99.11%</td>
<td>97.64%</td>
<td>99.11%</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-11176</td>
<td>92.17%</td>
<td>99.11%</td>
<td>92.17%</td>
<td>99.11%</td>
</tr>
<tr>
<td>DF</td>
<td>2017-2636</td>
<td>95.96%</td>
<td>99.11%</td>
<td>95.96%</td>
<td>99.11%</td>
</tr>
<tr>
<td>DF</td>
<td>2017-7308</td>
<td>98.50%</td>
<td>99.11%</td>
<td>98.50%</td>
<td>99.11%</td>
</tr>
</tbody>
</table>

Evaluation. We evaluated Context Conservation on all
applicable vulnerabilities and compared it against the baseline.

Applicability. Applying Context Conservation to kernel ex-
ports 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.

7.2 Compositing Stabilization Techniques

The kernel heap exploit model and the evaluation result in-
dicate that each individual technique mitigates different un-
reliability factors to different degrees. As such, we further
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Idle Baseline</th>
<th>Real-World</th>
<th>Combo</th>
<th>Busy Baseline</th>
<th>Real-World</th>
<th>Combo</th>
<th>R.W. Tech</th>
<th>Combo Tech</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOB</td>
<td>2017-7533</td>
<td>20.43%</td>
<td>N/A</td>
<td>95.37% [+]+</td>
<td>7.86%</td>
<td>N/A</td>
<td>82.60% [+]+</td>
<td>N/A</td>
<td>④⑤⑥</td>
</tr>
<tr>
<td>OOB</td>
<td>2017-7184</td>
<td>33.51%</td>
<td>N/A</td>
<td>97.86% [+]+</td>
<td>5.37%</td>
<td>N/A</td>
<td>77.74% [+]+</td>
<td>N/A</td>
<td>④⑤⑥</td>
</tr>
<tr>
<td>OOB</td>
<td>2016-6187</td>
<td>32.89%</td>
<td>99.34% [+]+</td>
<td>99.89% [+]+</td>
<td>2.49%</td>
<td>0.46%</td>
<td>60.63% [+]+</td>
<td>N/A</td>
<td>①⑤⑥</td>
</tr>
<tr>
<td>OOB</td>
<td>2010-2959</td>
<td>22.09%</td>
<td>92.94% [+]+</td>
<td>99.57% [+]+</td>
<td>5.83%</td>
<td>88.54% [+]+</td>
<td>95.09% [+]+</td>
<td>N/A</td>
<td>③⑤⑥</td>
</tr>
<tr>
<td>OOB</td>
<td>2017-7308</td>
<td>0.04%</td>
<td>67.92% [+]+</td>
<td>84.90% [+]+</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.76% [+]</td>
<td>N/A</td>
<td>⑤④⑤⑥</td>
</tr>
<tr>
<td>UAF</td>
<td>2018-6555</td>
<td>98.80%</td>
<td>55.00% [+]+</td>
<td>100.00% [+]+</td>
<td>61.92%</td>
<td>58.18%</td>
<td>100.00% [+]+</td>
<td>100.00% [+]+</td>
<td>③④⑥</td>
</tr>
<tr>
<td>UAF</td>
<td>2016-8655</td>
<td>96.06%</td>
<td>84.54% [+]+</td>
<td>99.96% [+]+</td>
<td>0.28%</td>
<td>83.28%</td>
<td>99.54% [+]+</td>
<td>N/A</td>
<td>③④⑥</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-15649</td>
<td>13.82%</td>
<td>99.68% [+]+</td>
<td>99.96% [+]+</td>
<td>18.34%</td>
<td>98.78%</td>
<td>99.74% [+]+</td>
<td>N/A</td>
<td>④④⑥</td>
</tr>
<tr>
<td>UAF</td>
<td>2016-4557</td>
<td>99.34%</td>
<td>N/A</td>
<td>100.00% [+]+</td>
<td>22.80%</td>
<td>N/A</td>
<td>99.09% [+]+</td>
<td>N/A</td>
<td>④④⑥</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-8824</td>
<td>97.64%</td>
<td>65.16% [+]+</td>
<td>99.56% [+]+</td>
<td>93.70%</td>
<td>61.32%</td>
<td>99.68% [+]+</td>
<td>①⑥⑥</td>
<td></td>
</tr>
<tr>
<td>UAF</td>
<td>2016-0728</td>
<td>1.48%</td>
<td>76.68% [+]+</td>
<td>100.00% [+]+</td>
<td>0.14%</td>
<td>34.22%</td>
<td>99.48% [+]+</td>
<td>④</td>
<td></td>
</tr>
<tr>
<td>UAF</td>
<td>2017-10661</td>
<td>31.98%</td>
<td>44.96% [+]+</td>
<td>95.96% [+]+</td>
<td>0.18%</td>
<td>4.60%</td>
<td>88.18% [+]+</td>
<td>①</td>
<td></td>
</tr>
<tr>
<td>UAF</td>
<td>2016-10150</td>
<td>0.74%</td>
<td>72.56% [+]+</td>
<td>99.88% [+]+</td>
<td>47.62%</td>
<td>81.20%</td>
<td>88.48% [+]+</td>
<td>⑤④⑥</td>
<td></td>
</tr>
<tr>
<td>UAF</td>
<td>2017-11176</td>
<td>92.17%</td>
<td>98.23% [+]+</td>
<td>100.00% [+]+</td>
<td>2.06%</td>
<td>51.31%</td>
<td>96.57% [+]+</td>
<td>④④⑥</td>
<td></td>
</tr>
<tr>
<td>DF</td>
<td>2017-2636</td>
<td>95.86%</td>
<td>N/A</td>
<td>98.80% [+]+</td>
<td>93.26%</td>
<td>N/A</td>
<td>98.80% [+]+</td>
<td>N/A</td>
<td>④④⑥</td>
</tr>
<tr>
<td>DF</td>
<td>2017-6074</td>
<td>98.50%</td>
<td>99.14% [+]+</td>
<td>92.14% [+]+</td>
<td>66.16%</td>
<td>89.76%</td>
<td>87.66% [+]+</td>
<td>③④⑥</td>
<td></td>
</tr>
<tr>
<td>DF</td>
<td>2017-8890</td>
<td>51.96%</td>
<td>N/A</td>
<td>80.92% [+]+</td>
<td>6.14%</td>
<td>N/A</td>
<td>72.04% [+]+</td>
<td>N/A</td>
<td>④④⑥</td>
</tr>
</tbody>
</table>

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
the baseline exploit against CVE-2017-6074, especially in the id
e 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, improv-
ing the baseline exploit reliability to some extent. This ex-
plains 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 Sys-
tem [14], or CVSS score in short), exploit reliability serves
as an important factor. Misunderstanding exploit reliability
may mislead developers by presenting a severe vulnerab-
ility 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 un-
derstand the kernel exploit reliability issue, helping them to
correctly prioritize patches to vulnerabilities.

Real-world Systems. We used two system workloads,
manifested by idle and busy, to demonstrate reliability variation
under different workloads for each stabilization technique.
Such a setup does not reflect the reliability of each stabi-
lization 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 Sys-
tems. This paper focuses on Linux operating systems, aim-
ing to understand the kernel heap exploit reliability issues
rooted in Linux kernel. However, kernel heap exploit reli-
ability 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. An-
other 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 re-
liability. 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 re-
sult 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. More-
ever, 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 col-
collection of research works on kernel exploitation. From tech-
nical perspectives, kernel exploitability assessment research
could be categorized into three sub-areas below.

The first research area is to explore the capability of vul-
nabilities. FUZE [43] searches for new use sites of UAF
vulnerabilities using under-context fuzzing and further em-
employs symbolic execution to identify exploitable primitives
implied by the new use sites. KOOBE [20] retrieves the cor-
ruption 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 primit-
ives. 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 manipulat-
ing the layout of kernel heap by using the kernel objects in
the database.

The third research area is to bypass mitigations in the ker-
nel. ret2dir [30] utilizes physical memory mapped to kernel
space for payload injection. KEPLER [42] uses communica-
tion channels between kernel space and user space to leak
stack canary and inject payload to kernel stack for ROP pro-
gramming. 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, re-
searchers have explored exploit generation approaches.

Brumley et al. [17, 19] proposed preconditioned symbolic
execution to automatically generate exploits for stack over-
flow and format string vulnerabilities. Bao et al. [18] de-
veloped shellcode layout remediation and path kneading ap-
proaches that automatically transplant existing shellcode to
the target vulnerability. The Shellphish team designed Pov-
Fuzzer and Rex to turn crashes into exploits [36, 37, 38]. Pov-
Fuzzer 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 ge-
etic 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 pro-
gram 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 stabil-
zation 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
c 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 expe-
rienced security researchers better understand stabilization
techniques. In addition to the study, this research also pro-
poses a new technique to improve exploit reliability. We show
that by combining the new technique with existing stabiliza-
tion 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 Ad-
vanced 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 Re-
search (ONR) under Grant No. N00014-20-1-2008, the Na-
tional 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


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: 1) The public exploits demonstrate the potential of obtaining control flow hijacking primitive, which indicates successful exploitation. 2) 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. 3) 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 any 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>CVE</th>
<th>Baseline</th>
<th>Defragmentation</th>
<th>Heap Grooming</th>
<th>Single-Thread Heap Spray</th>
<th>Multi-Process Heap Spray</th>
<th>CPU Pinning</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Idle State</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOBY</td>
<td>2017-7533</td>
<td>24.72%</td>
<td>52.42% [-]</td>
<td>N/A</td>
<td>99.36% [+]</td>
<td>99.62% [+]</td>
<td>36.42% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-7184</td>
<td>30.72%</td>
<td>99.78% [+1]</td>
<td>100.00% [-]</td>
<td>46.74% [-]</td>
<td>96.96% [-]</td>
<td>64.12% [-]</td>
</tr>
<tr>
<td></td>
<td>2016-6187</td>
<td>51.86%</td>
<td>N/A</td>
<td>N/A</td>
<td>73.56% [-]</td>
<td>99.87% [-]</td>
<td>99.56% [-]</td>
</tr>
<tr>
<td></td>
<td>2010-2959</td>
<td>15.48%</td>
<td>52.42% [+]</td>
<td>N/A</td>
<td>65.22% [-]</td>
<td>66.26% [-]</td>
<td>21.52% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-7388</td>
<td>0.02%</td>
<td>0.00% [-]</td>
<td>N/A</td>
<td>0.00% [-]</td>
<td>4.64% [-]</td>
<td>0.04% [-]</td>
</tr>
<tr>
<td>UAF</td>
<td>2018-6555</td>
<td>96.52%</td>
<td>82.24% [-]</td>
<td>N/A</td>
<td>100.00% [-]</td>
<td>100.00% [-]</td>
<td>98.56% [-]</td>
</tr>
<tr>
<td></td>
<td>2016-8655</td>
<td>97.18%</td>
<td>96.08% [-]</td>
<td>N/A</td>
<td>99.74% [-]</td>
<td>99.80% [-]</td>
<td>99.56% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-15649</td>
<td>17.02%</td>
<td>17.98% [-]</td>
<td>N/A</td>
<td>73.56% [-]</td>
<td>99.86% [-]</td>
<td>99.98% [-]</td>
</tr>
<tr>
<td></td>
<td>2016-4557</td>
<td>98.76%</td>
<td>97.94% [-]</td>
<td>N/A</td>
<td>99.00% [-]</td>
<td>99.98% [-]</td>
<td>99.80% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-8824</td>
<td>97.96%</td>
<td>0.04% [-]</td>
<td>N/A</td>
<td>97.46% [-]</td>
<td>99.24% [-]</td>
<td>99.98% [-]</td>
</tr>
<tr>
<td></td>
<td>2016-0728</td>
<td>1.74%</td>
<td>2.32% [-]</td>
<td>N/A</td>
<td>82.20% [-]</td>
<td>100.00% [-]</td>
<td>2.68% [-]</td>
</tr>
<tr>
<td>UAF</td>
<td>2017-10661</td>
<td>18.80%</td>
<td>20.00% [+]</td>
<td>N/A</td>
<td>35.38% [-]</td>
<td>80.86% [-]</td>
<td>73.66% [-]</td>
</tr>
<tr>
<td></td>
<td>2016-10150</td>
<td>0.00%</td>
<td>0.00% [-]</td>
<td>N/A</td>
<td>0.00% [-]</td>
<td>99.64% [-]</td>
<td>99.98% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-11176</td>
<td>96.08%</td>
<td>7.36% [-]</td>
<td>N/A</td>
<td>99.96% [-]</td>
<td>99.10% [-]</td>
<td>88.70% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-2636</td>
<td>99.42%</td>
<td>14.04% [-]</td>
<td>N/A</td>
<td>99.86% [-]</td>
<td>99.72% [-]</td>
<td>99.70% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-6074</td>
<td>99.98%</td>
<td>95.14% [-]</td>
<td>N/A</td>
<td>99.86% [-]</td>
<td>94.74% [-]</td>
<td>99.88% [-]</td>
</tr>
<tr>
<td></td>
<td>2017-8890</td>
<td>56.44%</td>
<td>0.30% [-]</td>
<td>N/A</td>
<td>58.52% [-]</td>
<td>75.66% [-]</td>
<td>76.98% [-]</td>
</tr>
</tbody>
</table>

| In Busy State |            |          |                 |               |                          |                          |             |
| OOBY | 2017-7533   | 5.56%    | 10.52% [-]      | N/A           | 71.58% [-]              | 97.74% [-]               | 13.14% [-]  |
|      | 2017-7184   | 9.78%    | 51.42% [-]      | 64.98% [-]   | 44.96% [-]              | 89.32% [-]               | 9.48% [-]   |
|      | 2016-6187   | 5.36%    | 53.78% [-]      | N/A           | N/A                     | N/A                      | 10.78% [-]  |
|      | 2010-2959   | 9.96%    | 47.54% [-]      | 32.84% [-]   | 36.04% [-]              | 65.72% [-]               | 19.06% [-]  |
|      | 2017-7388   | 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% [-]  |
|      | 2016-8655   | 42.20%   | 22.2% [-]       | N/A           | 33.56% [-]              | 99.76% [-]               | 64.64% [-]  |
|      | 2017-15649  | 10.32%   | 10.18% [-]      | N/A           | 38.50% [-]              | 99.82% [-]               | 98.72% [-]  |
|      | 2016-4557   | 51.82%   | 59.92% [-]      | N/A           | 97.48% [-]              | 99.84% [-]               | 60.14% [-]  |
|      | 2017-8824   | 96.66%   | 0.00% [-]       | N/A           | 86.56% [-]              | 98.46% [-]               | 100.00% [-] |
|      | 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% [-]   |
|      | 2016-10150  | 0.00%    | 0.00% [-]       | N/A           | 43.70% [-]              | 49.30% [-]               | 97.88% [-]  |
|      | 2017-11176  | 1.90%    | 2.32% [-]       | N/A           | 41.52% [-]              | 94.24% [-]               | 3.72% [-]   |
|      | 2017-2636   | 96.46%   | 24.38% [-]      | N/A           | 99.00% [-]              | 99.34% [-]               | 99.56% [-]  |
|      | 2017-6074   | 74.09%   | 62.48% [-]      | N/A           | 75.40% [-]              | 93.04% [-]               | 78.22% [-]  |
|      | 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.
Table 8: Performance of stabilization techniques under the configuration of 2 CPUs and 4G memory.

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