KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities

Wei Wu, *Institute of Information Engineering, Chinese Academy of Sciences; Pennsylvania State University; School of Cybersecurity, University of Chinese Academy of Sciences;* Yueqi Chen and Xinyu Xing, *Pennsylvania State University;* Wei Zou, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cybersecurity, University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities

Wei Wu[1,2,3⋆], Yueqi Chen[2], Xinyu Xing[2], and Wei Zou[1,3]

[1]{CAS-KLONAT[†], BKLONSPT[‡]}, Institute of Information Engineering, Chinese Academy of Sciences, China

[2]College of Information Sciences and Technology, Pennsylvania State University, USA

[3]School of Cyber Security, University of Chinese Academy of Sciences, China
{wuwei, zouwei}@iie.ac.cn, {yxc431, xxing}@ist.psu.edu

Abstract

Automatic exploit generation is a challenging problem. A challenging part of the task is to connect an identified exploitable state (exploit primitive) to triggering execution of code-reuse (*e.g.,* ROP) payload. A control-flow hijacking primitive is one of the most common capabilities for exploitation. However, due to the challenges of widely deployed exploit mitigations, pitfalls along an exploit path, and ill-suited primitives, it is difficult to even manually craft an exploit with a control-flow hijacking primitive for an off-the-shelf modern Linux kernel. We propose KEPLER to facilitate exploit generation by automatically generating a "single-shot" exploitation chain. KEPLER accepts as input a control-flow hijacking primitive and bootstraps any kernel ROP payload by symbolically stitching an exploitation chain taking advantage of prevalent kernel coding style and corresponding gadgets. Comparisons with previous automatic exploit generation techniques and previous kernel exploit techniques show KEPLER effectively facilitates evaluation of control-flow hijacking primitives in the Linux kernel.

1 Introduction

Software bugs may have extremely serious consequences, especially for the OS kernel, where they could be fatal to the reliability and security of the entire OS because of the higher privilege that the kernel resides in and the abundance of hardware resources that the kernel has direct control of. Kernel bugs can lead to data leakage, privilege escalation and even persistant attacks [33]. One straightforward solution to minimize consequences of kernel bugs is to immediately patch all of the kernel bugs reported via mail lists and kernel fuzzers [72] [58] [52] [37]. In practice, considering the lack of manpower to patch all bugs timely, vendors typically prioritize their efforts to patch the bugs with more severe security implications after assessing their exploitability. With the deployment of various kernel mitigations, the exploitability of bugs has been obviously weakened but still hard to decide. Despite the undecidability of the general exploitability problem, sometimes a carefully crafted exploit could serve as a constructive proof of exploitability.

Capable of proving exploitability by generating working exploits from a vulnerability Proof-of-Concepts (PoC), automatic exploit generation is a preferred choice for exploitability assessment because its soundness and efficiency [7] [3] [9] [64] [5] [55] [30] [75]. More importantly, automatically generating concrete exploits could not only help exploitability evaluation, but also let a user to gain advantages in adversarial settings (e.g. Capture-The-Flag competitions) by scoring fast. Last but not least, these generated exploits could potentially help defender-side to evaluate the effectiveness of proposals of new kernel mitigation.

The common workflow of automatic exploit generation systems are similar. In general we can divide them into the following two steps: ❶*exploit primitive identification* and ❷*exploit primitive evaluation*. In the first step, they search for pre-defined exploit primitive ("exploitable" states) based on the crashing path triggered by a PoC input. In the second step, after pinpointing an exploit primitive, they add exploit constraints and perform *constraint solving* to generate a concrete input to exercise a predefined exploit technique (*e.g.,* `ret2libc` attack).

However, in order to generate working exploit for a control-flow hijacking primitive, there remains to be the following three challenges in the process of *exploit primitive evaluation* which limits the capability of automatic exploit generation techniques to target a complex real-world system such as the Linux kernel.

Challenge 1, exploit mitigation. Exploit mitigations are designed and introduced to reduce attack surface and raise the bar of exploitation. For a modern Linux kernel, many new hardware features [38] [11], compiler-assisted instrumentation [22] [40], sensitive data objects protection [12] [65] [21] [13] and virtualization-based hypervisor [49] [51] have

⋆The main part of the work was done while studying at Pennsylvania State University.

†Key Laboratory of Network Assessment Technology, CAS

‡Beijing Key Laboratory of Network Security and Protection Technology

been introduced as exploit mitigations. As a consequence, many kernel exploit techniques are no longer effective [36] [61] [41] [39] [77], despite the fact that heavier enforcement such as control-flow integrity (`CFI`) [2] [16] [79] [78] is still *not* widely-adopted by major Linux releases perhaps due to performance concerns.

Challenge 2, exploit path pitfall. Side effects of exploit primitives could terminate exploitation in middle. Memory corruption, occurred along with an exploit primitive, can frustrate the attempt to trigger the primitive the second time, because an *exploit path* could unavoidably contain instructions triggering an unexpected termination of exploitation. Such termination can be a kernel panic of invalid memory access or an infinite loop in a kernel thread.

Challenge 3, ill-suited exploit primitive. Lack of stack pivoting gadget [54] which is a vital step to perform `ROP` attack and insufficient control over general registers can make an exploit primitive ill-suited. Although some strong primitives have been proven exploitable and even Turing complete [35], there is still a gap between ill-suited exploit primitive and the requirement of mounting a certain exploit technique.

Considering the three challenges, it could be quite difficult to even manually craft an exploit with a control flow hijack primitive. To address the above challenges, we propose KEPLER, an exploit primitive evaluation framework for real-world Linux kernel vulnerabilities. KEPLER employs a novel exploit technique designed for Linux kernel which reduces exploitation of a control-flow hijacking primitive to constructing a classical overflow in kernel stack. The exploit technique exposes less constraints over the quality of an exploit primitive and availability of stack pivoting gadget than previous exploit techniques and could still bypass currently widely deployed kernel mitigations while previous approches could not. Starting from a possibly ill-suited *control-flow hijacking primitive* (`CFHP`), KEPLER overcomes the lack of stack pivoting gadget and manages to build a "single-shot" exploitation chain to bootstrap existing *Turing complete* exploit techniques such as return oriented programming (`ROP`) [56], with the ability to bypass mainstream kernel mitigations as well as detouring exploit path pitfalls.

To achieve this, KEPLER leverages a carefully designed code-reuse template for control-flow hijacking primitives to bypass widely-deployed mitigations in Linux kernel. KEPLER first enhances an exploit primitive to satisfy a minimal requirement of controlling dual registers (*e.g.,* `rip` and `rdi` for `x86-64`) at the same time. Starting from the primitive, KEPLER generates an exploit which sequentially executes a chain of five gadgets. The design of the code-reuse template involves several insights about Linux kernel coding style. Specifically, KEPLER reuses those stack-based invocations of kernel I/O channel functions to leak and smash kernel stack of current process and execute arbitrary user-supplied `ROP` payload. KEPLER leverages *blooming gadget* to enhance control over necessary registers for a control-flow hijacking primitive. KEPLER uses a *bridging gadget* to combine the practice of leaking kernel stack canary and smashing kernel stack into a single shot and thus prevent unexpected kernel panic.

To generate exploits for each `CFHP` against arbitrary kernel binary, KEPLER operates in the following two phase: first, KEPLER statically analyzes the kernel binary for five categories of candidate gadgets. Then KEPLER starts kernel symbolic execution from the `CFHP`, and performs a *Depth First Search* (DFS) based gadget stitching algorithm over candidate gadgets.

Our evaluation of KEPLER shows that it is powerful for exploit primitive evaluation. To highlight the effectiveness of KEPLER, we compare KEPLER with existing exploit hardening/generation tools (*e.g.,* `Q` [60], `fuze` [75]) and KEPLER outperforms all of them in terms of generating effective kernel exploits under modern mitigation settings in Linux kernel.

This research work makes the following contribution:

- **Kernel single-shot exploitation**. We present a code-reuse exploit technique which converts a single ill-suited control-flow hijacking primitive into arbitrary `ROP` payload execution under various constraints posed by modern Linux kernel mitigations and the primitive itself. The proposed technique exploits prevalent kernel coding style and corresponding gadgets and thus is hard to defeat. Our approach to calculate exploitation chain is automatable because the gadget stitching problem could be cast as a search problem over a search space of reasonable size. In addition, the "single-shot"[1] nature of this technique makes it suitable for the vulnerabilities prone to unexpected termination because it avoids stressing a control-flow hijacking primitive for multiple times.

- **Semi-automatic exploit generator for Linux kernel**. We implement KEPLER using a set of tools including `IDA SDK`, `QEMU`/`KVM` and `angr`. Starting from a user-supplied control flow hijack primitive, KEPLER analyzes the Linux kernel binary, tracks down useful kernel gadgets, and automatically generates many gadget chains for launching "single-shot" exploitation and bypassing kernel mitigations. It requires no kernel source code and can be applied to stripped kernel images. KEPLER applies to modern Linux kernels; our evaluation uses version 4.15.0 which was the latest as of the time of our evaluation.

- **Practical impacts**. We systematically evaluate the effectiveness and efficiency of KEPLER using 16 real-world kernel vulnerabilities and 3 recently-released CTF challenges. Given a kernel control-flow hijacking primitive, we show that KEPLER generally could generate tens of

[1]The proposed exploit technique requires a lot of analysis effort, but with respect to the precondition for launching the attack, it requires only a single control-flow hijacking primitive.

thousands of distinct exploitation chains with the ability to bypass kernel mitigations and perform successful exploitation. We show that KEPLER can output the first working exploit for a kernel vulnerability in less than about 50 wall-clock minutes.

2 Background and Related Work

Exploit primitives [6] [47] [55] are machine states that violate security policies at various level and indicate an attacker could get extra capabilities beyond the normal functionality provided by the original program. A `control-flow hijacking primitive` (`CFHP`) is a machine state that potentially deviates from the legal control-flow graph. In the context of symbolic analysis, a control-flow hijacking primitive is usually identified by applying a heuristic which queries the backend constraint solver to check whether the number of possible control flow jump target is beyond a threshold when the control flow jump target contains symbolic bytes. An arbitrary memory write primitive is a machine state that an attacker could modify arbitrary kernel memory on his will. Similarly, an arbitrary memory leak primitive is a machine state which allows an attacker to dump data content at arbitrary kernel address. Sometimes the primitive does not allow an attacker to modify/leak data at arbitray kernel address (*e.g.,* a stack info leak which only dump several bytes on kernel stack [68]), they are referred as restricted memory write/leak primitive.

As is described above, this research work mainly focuses on two aspects – ❶ facilitating exploit primitive evaluation for even ill-suited exploit primitives and bypassing widely-deployed kernel mitigation mechanisms by designing a new exploit technique. ❷ developing a tool to automate the newly proposed kernel exploitation approach. As a result, the works most relevant to ours include those pertaining to exploit primitive identification, exploit primitive evaluation and kernel exploit techniques/mitigations. In the following, we describe the existing works in these three directions and discuss their difference from ours.

2.1 Exploit Primitive Identification

To assist the process of finding a useful exploit primitive (*e.g.,* control-flow hijacking primitive and memory write/leak primitive), there is a rich collection of research works. For example, using preconditioned symbolic execution and concolic execution techniques, Brumley *et al.* develop `AEG` as well as `mayhem` to identify control-flow hijacking primitives for further exploitation. [3] [9] [7]. Shoshitaishvili *et al.* develop a cyber reasoning system `Mechanical Phish` [62]. It is built on angr [64] [69] [63] and performs fuzzing and symbolic tracing for the PoC to identify exploit primitives. To efficiently explore state space for exploit primitives in Linux kernel, Wu *et al.* propose an automatic technique that utilizes under-context fuzzing along with partial symbolic execution to explore `CFHP` and memory write primitive for UAF bugs [75]. To construct better exploit primitives with the capability of out-of-bound access, Heelan *et al.* utilize regression tests to obtain the knowledge of how to perform heap layout manipulation [30]. To obtain better exploit primitives for stack Use-Before-Initialization vulnerabilities, Lu *et al.* propose a deterministic stack spraying approach as well as an exhaustive memory spraying technique [46].

In this work, we do not focus on facilitating primitive identification. Rather, we assume an exploit primitive is already identified and our research endeavor centers around subsequent primitive evaluation phase.

2.2 Exploit Primitive Evaluation

In the primitive evaluation phase, a security analyst or an automatic exploit generation system tries suitable exploit techniques for the seemingly exploitable states identified before.

Initially, without considering Data Execution Prevention, such systems use straightforward techniques such as `ret2stack-shellcode` and `ret2libc` with a `CFHP` [3] [9] [62]. Taking $W \oplus X$ into account, Schwartz *et al.* propose Q [60] to facilitate exploitation with a `CFHP` by automatically constructing a `ROP` chain. Our work addresses the problem of `ROP` bootstrapping without stack pivoting gadget and is orthogonal to automatic `ROP` chaining techniques [33] [60] [66] [24] [57] because we do not tackle the problem of `ROP` payload construction.

With the facilitation of forward and backward taint analysis, Mothe *et al.* devise a technical approach to craft working exploits for simple vulnerabilities in user-mode applications [48]. Utilizing symbolic execution, Repel *et al.* craft exploits with single memory write primitive (*e.g.,* unsafe unlink and lookaside list corruption) for those heap overflow vulnerabilities residing in the userland applications [55]. To facilitate primitive evaluation of kernel Use-After-Free exploitation, Xu *et al.* propose two memory collision mechanisms [77] to unleash `CFHP`s.

Recently, some research works take `CFI` into consideration for primitive evaluation [29] [8] [59] [23] [31] [32] [35]. For example, Ispoglou *et al.* propose block oriented programming (`BOP`) [35] to facilitate evaluation of repeatable arbitrary memory write primitives by proving the Turing completeness under `CFI` along with common userspace mitigations. `BOP` assumes the existence of a dispatcher gadget. `BOP` also automates exploit generation. As a repeatable arbitrary memory write primitive is almost "god-mode" of kernel exploitation, our work facilitates primitive evaluation for those weaker exploit primitives (*e.g.,* non-repeatable and ill-suited `CFHP`) in real-world .

In this work, we also develop a tool for facilitating primitive evaluation. However, this research work is fundamentally different from the aforementioned works in at least one of the following aspects. First, without assuming a perfect exploit primitive (*e.g.,* unlimited number of invocations of an arbitrary memory write primitive), we can faciliate exploit primitive evaluation for those usually ignored exploit primitive (*e.g.,* ill-suited primitives and primitive that can only be triggered once). Second, rather than dealing with applications in the user space, our tool targets the Linux kernel where exploitation typically involves complicated operations and sophisticated security mitigation mechanisms are generally enabled. Third, rather than generating one single exploit for a target vulnerability, our tool automatically explores many possible exploitation chains and output various working exploits.

2.3 Kernel Exploit Techniques/mitigations

Initially, a `CFHP` in the kernel can directly execute shellcode in user-space because there is no isolation between user and kernel space (*e.g.,* `ret2usr`). Supervisor Mode Execution Prevention (`SMEP`) [38] prevents kernel from executing userspace code. An attacker can use code-reuse attack. To set stack pointer to controlled payload, she uses the prevalent "pivot-to-userspace" gadget to pivot stack to userspace [44]. With adoption of Supervisor Mode Access Prevention (`SMAP` [11]), an attacker can no longer rely on a fake stack in userspace because userspace memory access is forbidden except during I/O channel functions. Because there is usually none intra-kernel stack pivoting gadget for a Linux kernel, an attacker usually chooses to disable `SMAP` by flipping corresponding bits in the `cr4` register [41]. However, the "`cr4`-flipping" attack typically rely on double `CFHP`s [41] and not suitable for a *none re-triggerable* `CFHP`. In addition, a virtualization-based hypervisor can detect `cr4` register modification by inspecting a `vmexit` and thus mitigate such exploitation [49] [51]. `Ret2dir` [39] attack sprays the `physmap` region by calling syscall `mmap`, as the direct mapped physical memory is marked as executable previously, diverting a `CFHP` to land on physmap led to arbitary shellcode exectuion. However, with a kernel patch applied, these `physmap` pages are no longer executable.

To enforce CFI policy for the kernel, several kernel `CFI` solutions [16] [70] [26] have been proposed, however, these mitigations are not broadly adopted by major Linux release version such as CentOS, Ubuntu and Debian.

Data-only kernel exploit techniques directly use a memory write primitive to modify sensitive kernel data objects such as process credentials, page tables and virtual dynamic shared object (vdso) [36]. However, mitigations have been proposed [17] [67] and deployed [40] to prevent these low-hanging fruits.

Note some memory write primitive can be transformed to a control-flow hijacking primitive by overwriting and invoking a code pointer in kernel's data section or in the heap [19].

Kernel address space layout randomization (KASLR) is widely deployed in order to present the attacker an unpredictable attack surface. However, due to its lack of timely re-randomization and coarse-grained nature (only randomizing section base address), an attacker does not even need an arbitrary read primitive [45] [71] to bypass KASLR. With a hardware side channel [34] [28], he can infer the coarse memory layout without leaking any kernel memory content. Despite kernel Page Table Isolation (`KPTI` [13]) removes some side channels with extra overhead, he can also uses a restricted memory leak primitive to infer the coarse memory layout [68]. In the default setting of Linux kernel, knowing coarse memory layout is enough for various exploit techniques. (Kernel) Code diversification/randomization [14] [15] [27] [74] [42] [53] could significantly raise the bar of code-reuse exploitation by thwarting an attacker from locating useful gadget.

3 Assumptions and Threat Model

Our threat model consists of a modern Linux kernel protected by widely-deployed mitigations with a known vulnerability.

Mitigation setting. Similar to recent major Linux release versions, we make the following assumptions of kernel mitigations. A kernel has enabled `SMEP` and `SMAP` [11] protection to prevent direct userspace access in kernel execution. A kernel has enabled stack canary to protect return address over stack for all functions containing local variable [22]. A kernel has enabled protection to prevent direct modification of sensitive kernel data objects including process credential [40] and page table [17]. A kernel has enabled `KASLR`. A kernel has enabled `KPTI` [13] protection. A kernel has been protected by virtualization-based hypervisor which prevents unauthorized modification of `cr4` regsiter [49]. A kernel has set physmap pages as non-executable. A kernel has enabled `STATIC_USERMODEHELPER`. The option routes all `call_usermodehelper()` calls through a guard binary that can properly filter the requested userland programs to be run by the kernel [43]. However, A kernel does `not` enable a `CFI` enforcement such as `RAP` [70] because of performance concerns.
Available Exploit Primitives. We assume there is a PoC which triggers the vulnerability and leads to a control-flow hijacking primitive (`CFHP`). We assume the `CFHP` is already identified with the PoC through either manual analysis or dynamic analysis such as symbolic tracing, thus finding exploit primitives is orthogonal to our work and we can focus on evaluating the `CFHP`. We assume a restricted memory leak primitive to help infer coarse kernel memory layout (*e.g.,* getting the base address of code section `.text` and `physmap` region). We do `not` assume the existence of an arbitrary memory write primitive which could write to arbitrary kernel memory address. We do *not* assume the existence of an arbitrary memory leak primitive that can dump arbitrary kernel memory content. Under the threat model, the content in arbitrary memory address such as the stack canary value of arbitrary kernel thread usually remains secret because the coarse kernel memory layout information does not reveal the stack canary value of a kernel stack. We also assume the location of current kernel stack

remains secret although a restricted memory leak primitive might help leak a pointer to current stack under some specific vulnerability context.

4 Motivating Example

```
struct ip_mc_socklist {
  struct ip_mc_socklist *next_rcu;
  struct ip_mreqn multi;
  unsigned int sfmode;
  struct ip_sf_socklist *sflist;
  struct rcu_head rcu;
};
```

(a) Definition of struct `ip_mc_socklist`. Its first member `next_rcu` is unmanageable because the PoC uses a heap spray technique which does not allow us to control first `QWORD` of struct `ip_mc_socklist`.

```
1  void ip_mc_drop_socket(struct sock *sk){
2    struct inet_sock *inet = inet_sk(sk);
3    struct ip_mc_socklist *iml;
4    // inet->mc_list is a dangling pointer
5    while ((iml = inet->mc_list) != NULL) {
6      // iml is alias of the dangling pointer
7      inet->mc_list = iml->next_rcu;
8      // queuing a rcu_head for execution in
           the future
9      kfree_rcu(iml, rcu);
10   }
11 }
12 void rcu_reclaim(struct rcu_head *head){
13   head->func(head); // control-flow hijack
14 }
15 void rcu_do_batch(...){
16   struct rcu_head *next, *list;
17   while (list) {
18     next = list->next; // next rcu is
           unmanageable
19     rcu_reclaim(list);
20     list = next;
21   }
22 }
```

(b) Tailored source code pertaining to the `CFHP`. function `ip_mc_drop_socket` repeat invoking `kfree_rcu()` which queues a `rcu` task for asynchronous execution until `inet->mc_list` is `NULL`. The site pertaining to the `CFHP` is in function `rcu_reclaim`.

Table 1: A control-flow hijacking primitive in kernel UAF vulnerability CVE-2017-8890.

We illustrate the challenges in evaluating a `CFHP` on x86-64 with CVE-2017-8890 [50], a recent vulnerability in the Linux kernel.

4.1 Vulnerability and Exploit Primitive

The root cause of the bug is an Use-After-Free over an object `ip_mc_socklist` defined in Table 1a. As is shown in Table 1b, the UAF bug results in a dangling pointer `inet->mc_list` and `*iml` become an alias of the dangling pointer in line 5. In line 7, there is an UAF access by dereferencing `iml->next_rcu`. In line 9, `kfree_rcu(iml)` queues a callback denoted by `iml->rcu`. Function `rcu_do_batch` handles the previously queued callback when the CPU gets a chance to process the `rcu` callback list, thus triggers another UAF access to the `ip_mc_socklist` object in `rcu_reclaim()`. The loop from line 5 to line 10 will continue and add another callback if `iml->next_rcu` is not `NULL`.

The `CFHP` is due to an asynchronous `kfree_rcu` call over the dangling pointer `*iml`. To be specific, by manipulating the value in `iml->rcu_head` through a proper heap spray, a security analyst can get a `CFHP` later in `rcu_reclaim()` because function `kfree_rcu()` (line 9) is designed to queue a `rcu` callback denoted by `iml->rcu_head`. Function `rcu_do_batch` will be executed in the future, it will iterate over a list of `rcu_head` added by `kfree_rcu()`. The statement pertaining to the `CFHP` (line 13) allows the attacker to control `rip` (`head->func`) through heap spraying. At the time of the control-flow hijacking, register `rdi` (`head`) points to a controllable memory region.

4.2 Challenges of Crafting Working Exploit

However, developing an exploit with the aforementioned `CFHP` is quite difficult because of the following challenges.

4.2.1 Challenge 1: Exploit Mitigations

Widely deployed kernel mitigations frustrate a large amount of straight forward exploit techniques. With the presence of `SMEP/SMAP` protection, it is impossible to directly launch a traditional `ret2user`/`pivot2usr` attack. The `ret2dir` attack [39] is also not suitable because the `physmap` region is no longer executable [10]. With the write-protection over sensitive data such as process credential and page table , it is not possible to direct overwrite these data to escalate priviledge by converting the `CFHP` into a memory write primitive.

A security analyst may think of the "`cr4`-flipping" attack which usually requires two `CFHP`s: one for flipping the `cr4` register and the other to launch `ret2user`/`pivot2usr`. However, this is often infeasible because virtualization based hypervisor could easily detect the behavior of flipping `cr4` register by inspecting a `vmexit` [49] [51]. Even if there is not protection over `cr4` register, leveraging the "`cr4`-flipping" attack or a similar exploit technique relying multiple `CFHP` with this `CFHP` still faces the following challenge.

4.2.2 Challenge 2: Exploit Path Pitfall

Attempting to leverage the "`cr4`-flipping" exploit technique, a security analyst would expect two `CFHP`s to disable `SMEP/SMAP` and pivoting to userspace respectively. However, such a exploit technique could be imfeasible because of the exploit path pitfall after the first `CFHP`.

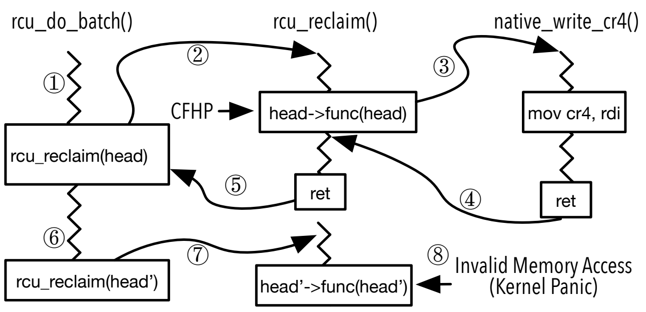


Figure 1: Demonstration of an exploit path pitfall in the motivating example. After applying the first CFHP to overwrite cr4 register with native_write_cr4(), rcu_reclaim(head′) is invoked but head′ is unmanageable and panics the kernel.

As is shown in Figure 1, after using the CFHP in rcu_reclaim to disable SMAP protection by invoking the function native_write_cr4() to zero the corresponding bits in cr4 register, an attacker encounters an unexpected termination: in previous execution, the loop from line 5 to line 10 in Table 1b queues another rcu callback denoted by head′ (iml->next_rcu ->rcu) which is not under our control and causes a kernel panic.

This kernel panic attributes to an invalid memory access. The heap spray technique used by the PoC does not allow an attacker to control first QWORD of iml (an ip_mc_socklist object) because the first QWORD of an freed chunk becomes heap-metadata, thus iml->next_rcu becomes unmanageable.

A straight-forward solution to tackle the invalid memory access is adding extra constraints to ensure all related memory accesses are valid. However, constraint solving for complex program usually incurs high cost (both cpu time and memory) [4] and could fail because of constraint conflicts [5]. Instead of devoting extra resource to handle these pitfalls, an ideal exploit path should effectively detour them. As a result of lacking control of iml->next_rcu, the exploit path pitfall can not be simply prevented by techniques like adding constraints over the sprayed data and thus unavoidably panic the kernel. Although it is possible to tackle the problem by further tuning the PoC [55] [75] and obtain a better exploit primitive, the showcased exploit path pitfall already hinders the evaluation of the CFHP.

In addition, an exploit path pitfall could also be attributed to un-successful heap spray. Due to the undeterminacy of kernel execution, there is not any heap spray technique with 100% success ratio. To get multiple CFHPs for a UAF vulnerability, an attacker may need to do multiple rounds of heap spraying and trigger a vulnerability multiple times, which could dramastically decrease the success ratio of the entire exploitation.

Even if two control-flow hijack primitive is available to the attacker, it could still be very difficult for him to bypass the mitigation combination of SMAP as well as virtualization-based hypervisor, because the attempt to modify cr4 register (in order to turn off SMAP and pivot kernel stack to userspace) could be easily prevented.

4.2.3 Challenge 3: Ill-suited Exploit Primitive

Facing the infeasibility of popular kernel exploit techniques, it is nature for a security analyst to use generic code-reuse technique such as ROP [56] as a second resort.

Stack pivoting is a vital step [54] for a ROP attack especially when an attacker does not control the contents on the stack (*e.g.,* the CFHP does not result from a stack overflow). In userspace, many heap exploits relying on a stack pivoting gadget (*e.g.,* function swapcontext() and setcontext() in libc) to bootstrap a ROP attack.

It is however difficult in the target kernel to pivot stack pointer to a memory region under our control with this CFHP. The reason behind is two-fold. First, as is mentioned before, we can not simply pivot to a userspace with a traditional gadget such as xchg eax,esp; ret because of SMAP. Second, there is not a suitable stack pivoting gadget in a Linux kernel binary for this CFHP. Considering register rdi points to controllable area with this primitive, it would be great to have a gadget to overwrite rsp with a controllable memory address, such as gadget with form xchg r**,rsp; ret, mov rsp,[r**]; jmp rxx and mov rsp,r**; ret for consecutive payload [39]. Unfortunately, similar traditional stack-pivoting gadget does not exist or contains unavoidable exploit path pitfall (*e.g.,* gadget xchg rsp, r14 ; jmp rsp could successfully pivot the stack pointer but inevitably panics the kernel) in our investigation across various modern linux kernel images.

Although previous works have demonstrated the power of code-reuse attack, mounting a traditional ROP attack for the CFHP seems surprisingly difficult because of the lack of stack pivoting gadget.

Without the capability of performing ROP attack, one may think of reusing other kernel functions. Unfortunately, there is also a problem of insufficient control over general registers because only rdi points to a memory region under control and other registers are not in control initially. We need to enhance this CFHP by controlling more general registers and perform subsequent exploitation.

5 Overview

To tackle the three challenges exposed by the motivating example, a security analyst needs to design a new exploit technique to turn a CFHP into a more exploit-friendly machine state based on the vulnerability context and his prior experience. Due to the lack of a ready-to-use exploit technique, he could expect a lot of debugging and manual efforts to explore possible exploit paths and improve his prototype exploit during the exploit development process and such practice could be extremely time-consuming and even fruitless.

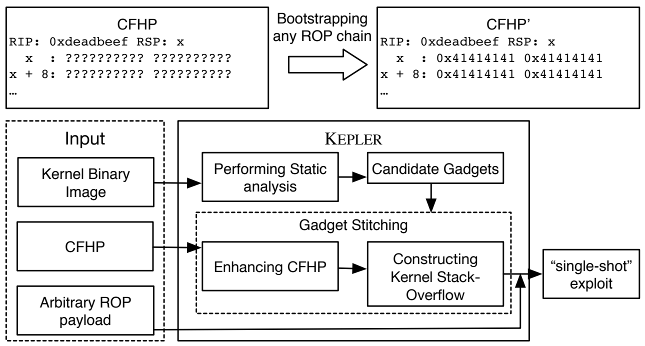
In the following, we discuss the considerations that go into the design of KEPLER as well as the high level design of this framework to facilitate CFHP evaluation.

5.1 Requirements for Design

To support evaluation of a CFHP with working exploits, KEPLER should receive as input a state representing a CFHP and it should be able to find an exploit path towards priviledge escalation and output corresponding exploit. To achieve the above ultimate goal, KEPLER should adopt an exploit technique which satisfies the following requirements.

First, an exploit technique is able to bypass all the widely-deployed mitigations enabled in the threat model. *Second*, taking potential exploit path pitfalls into consideration, an exploit technique should depend on only one control-flow hijacking primitive and detour these pitfalls to prevent an unexpected termination and make exploitation more reliable. The form of an exploit technique would be ideally similar to a "magic gadget[2]" which is previously mentioned in user-space exploitation [18], especially in the context of adversarial scenarios like Capture-The-Flag cyber competition. *Third*, an exploit technique should benefit from time-tested exploit technique such as ROP by efficiently bootstrapping traditional code-reuse attack in absence of stack pivoting gadget with an ill-suited CFHP. *Last but not least*, an exploit technique should be suitable for the automation framework. On one hand, it should be hard-to-defeat and not depend on any special code or feature which could be easily eliminated. On the other hand, the exploit generation phase should be easily automated - it should be a well-defined search problem over a search space of reasonable size.

5.2 High Level Design


```
CFHP
RIP: 0xdeadbeef RSP: x
  x  : ?????????? ??????????
x + 8: ?????????? ??????????
...

CFHP'
RIP: 0xdeadbeef RSP: x
  x  : 0x41414141 0x41414141
x + 8: 0x41414141 0x41414141
...
```

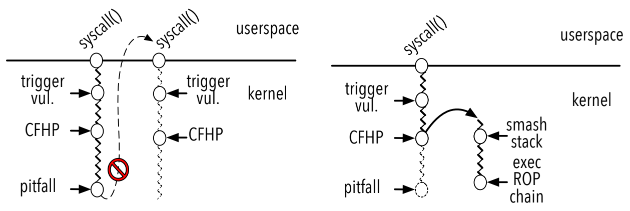
Figure 2: Overall of KEPLER's design.

[2]The term "magic gadget" means given a CFHP, one can instantly succeed in exploitation (*e.g.,* getting a shell) by diverting control-flow to such gadget, thus boost exploit development and gain advantages in a game.

To satisfy the requirements mentioned above, we design KEPLER to facilitate exploit primitive evaluation. KEPLER automatically generates an exploitation chain to *bootstrap* any kernel ROP chain with a single CFHP through "single-shot" exploitation.

Figure 2 shows how KEPLER automates the analysis task necessary to leverage a CFHP to produce an exploit in the presence of aforementioned challenges. Given a kernel state snapshot representing the CFHP, KEPLER enhances its power to construct a kernel stack-overflow by symbolically stitching several types of candidate gadget identified by static analysis on the kernel binary image.

As is mentioned before, our basic idea is to *bootstrap* a traditional ROP attack with a CFHP in Linux kernel. At the high level, we achieve this by a "single-shot" exploitation chain which transforms a function pointer corruption based primitive (CFHP) into a stack-overflow based primitive (CFHP') as is shown in Figure 2.



(a) Exploitation by envoking a control-flow hijacking primitive twice.

(b) Exploitation with a single control flow hijack primitive.

Figure 3: A comparison of two exploitation approaches; a known approach triggers a vulnerability twice but blocked by an exploit path pitfall and the other triggers the vulnerability only once.

Although there is not any gadget in Linux kernel which allows an attacker to directly escalate priviledge, The proposed "single-shot" exploitation is similar to "magic gadget" mentioned above in a sense that it only requires a single CFHP and could reliably achieve the goal of arbitrary code execution in kernel context. To be specific, as is shown in Figure 3b, the "single-shot" exploitation chain could finish exploitation with a single CFHP and thus is able to circumvent an exploit path pitfall after the return of CFHP which could cause an unexpected termination otherwise. We can benefit from a stack-overflow based CFHP because it allows us to place arbitrary ROP payload on current kernel stack without any stack pivoting gadget. Given the scarcity (or non-existence) of intra-kernel stack pivoting gadget, we argue that constructing a kernel stack overflow is the most generic approach to perform a kernel ROP attack.

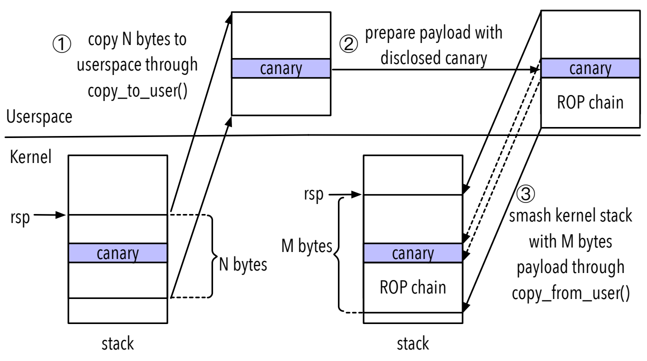

① copy N bytes to userspace through copy_to_user()
② prepare payload with disclosed canary
③ smash kernel stack with M bytes payload through copy_from_user()
Userspace
Kernel
rsp
canary
N bytes
M bytes
ROP chain
stack

Figure 4: An overview of "single-shot" exploitation which discloses kernel stack canary and then smashes the kernel stack with arbitrary user-supplied ROP payload.

6 Design

In this section, we describe the exploit technique adopted by KEPLER and the insights behind the exploit tehcnique. As is mentioned before, KEPLER uses a `CFHP` to construct a stack overflow and bootstraps arbitrary `ROP` payload.

Our "single-shot" exploitation technique builds on two key ideas of *breaking isolation with I/O functions* and *improving exploit success ratio with a single `CFHP`*.

First, we can break isolation between kernel-space and user-space by abusing kernel I/O functions. The insight behind is such data channels are born to bypass `SMAP` which prohibits user-space access because `SMAP` is explicitly and temporarily disabled during execution of these functions. Figure 4 illustates a practice of reusing I/O functions to construct kernel stack overflow by first leaking kernel stack canary with `copy_to_user` and then smashing kernel stack with `copy_from_user`.

Second, "single-shot" exploitation can be achieved through stitching various kernel function gadgets. We can enhance register control for a `CFHP` with *blooming gadget* - a prevalent family of function gadgets in Linux kernel. We can detour exploit path pitfalls with a *bridging gadget*.

6.1 Constructing Stack Overflow

There is a family of prevalent stack smashing gadgets inside Linux kernel, we observe they could greatly aid constructing stack-overflow via taking intrinsic *short return path* triggered by a page fault. However, such gadgets can not be directly used because initial `CFHP` does not have enough register control and the presence of a stack canary. We address the two problem in Section 6.2 and 6.3.

6.1.1 Looking into Stack-Smashing Gadget

We present stack-smashing gadgets which relies on functions that serve as data channel between user-space and kernel-space. The gadget could aid exploitation by transporting payload of arbitrary length from user-space to kernel stack, without assuming the stack location is already known.

```
1 static long bsg_ioctl(struct file *file, unsigned
      int cmd, unsigned long arg){
2   struct sg_io_v4 hdr;
3   ...
4   if (copy_from_user(&hdr, uarg, sizeof(hdr)))
5     return -EFAULT; // short return
6   ...
7 }
```

(a) Source code.

```
1  ...
2    mov rdi,rsp
3    call <_copy_from_user>
4    test rax,rax
5    je 0xffffffff813d6ce4
6    mov rax,0xfffffffffffffff2
7    jmp <epilogue>
8  ...
9  <epilogue>:
10   mov rcx,QWORD PTR [rsp+0xa0]
11   xor rcx,QWORD PTR gs:0x28
12   jne <__stack_chk_fail>
13   add rsp,0xa8
14   pop rbx
15   ret
```

(b) Assembly code.

Table 2: The kernel code fragment of an stack-smashing gadget that could smash kernel stack with carefully crafted payload.

As is named after, `copy_from_user(void* dst, void* src, unsigned long length)`[3] is a heavily used I/O kernel function which migrates data from the user to kernel space. Recall that `SMAP` prevents kernel code from accessing user-space address, and to temporarily bypass the restriction, `copy_from_user` uses a special instruction `STAC` to set `AC` flag in `EFLAGS` register before accessing user-space memory, thus allows the subsequent instructions to explicitly access user-space memory. Once the copy is finished, instruction `CLAC` is executed to re-enable `SMAP`.

As is specified in Linux kernel implementation, the function `copy_from_user()` takes as input three arguments - `dst`, `src` and length - which indicate the destination, source and length of the data that need to be copied from the user to kernel space.

From the perspective of an attacker, a kernel stack overflow could be caused if he lets the `CFHP` jump to the site right before the invocation of `copy_from_user` (line 2 in Table 2b) and the machine state satisfies the following three requirements:

[3]The security of `copy_from_user` has been improved by adding extra checks during the development of Linux Kernel. For example, upon failure during copy, set the `dst` memory region after the successfully copied bytes to zero to prevent uninitialized use.

❶ parameter `dst` (*e.g.,* `rdi`) points to current kernel stack, ❷ parameter `src` (*e.g.,* `rsi`) points to any user-space address so that its content is controllable by an attacker, ❸ parameter `length` (*e.g.,* `rdx`) is greater than the size of current stack frame to cause a kernel stack overflow.

An interesting observation is that most (91% in Linux 4.15) invocations of this function set destination `dst` to address of a variable on kernel stack and thus will copy user data into a kernel stack. If control-flow is hijacked to a invoking site of this function (*e.g.,* line 2 in Table 2b), an attacker could abuse such coding style to satisfy the requirement ❶ above because the code snippet help set `rdi` to current stack frame. However requirements ❷ and ❸ are still waiting to be satisfied given the initial `CFHP` does not imply any control over register `rdi` and register `rsi`. We will address this issue with blooming gadget in Section 6.3.

6.1.2 Choosing Short Return Path

The prevalent error handling code which is introduced to make kernel code more robust also provides a short return path for an attacker. An attacker could benefit from such a short return path after overflowing current stack frame.

As is depicted in line 4-5 in Table 2a, function `bsg_ioctl` will directly return if return value of `copy_from_user` is *not* zero. Our statistic indicates the function `copy_from_user()` has been invoked at 671 sites in Linux 4.15. Among all these invocation sites, more than 99% contain the fault handling implementation.

The insight of prioritizing short return path after stack smash is to prevent un-expected kernel panic as well as avoid the complexity of resolving extra data dependency in an error-prone and long normal return path of the function containing tens of basic blocks.

To take a short return path, `copy_from_user` must return a non-zero value as is shown in Table 2a. Reviewing the source code of kernel function `copy_from_user`, we have identified 3 different situations which will force the function to return a non-zero value, 1) incurring an integer overflow when calculating *src+length*, 2) neither `src+length` nor `src` residing in user-space, 3) encountering an unresolvable page fault during copy. For the first two situations, the function `copy_from_user()` performs sanity check and returns a non-zero value without actually copying data to the kernel. For the last situation, the function migrates data to the kernel and pads with zeros the bytes failing to be copied.

6.1.3 Triggering Page Fault during `copy_from_user`

To force `copy_from_user` returning a non-zero value as well as successfully copying the `ROP` payload from user-space to kernel stack, we trigger page fault after copying enough payload according to the last condition described above.

We illustrate a representative example in Figure 5. We map two adjacent pages ($p1$ and $p2$) in the user-space and

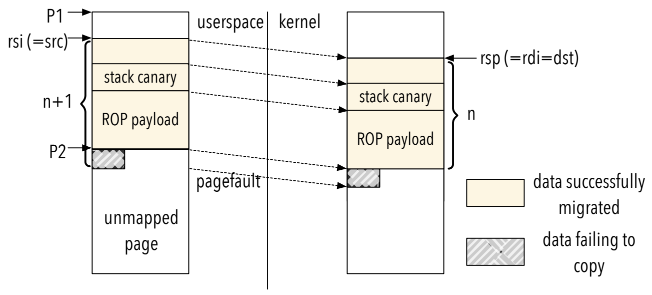


Figure 5: An example where `copy_from_user` triggers a page fault when copying user data to kernel stack.

then unmap the second one. We fill the end of the page $p1$ with the actual payload including a stack canary and a `ROP` chain. We will discuss how to leak stack canary in Section 6.2. Through the technical approaches mentioned in Section 6.3, assume we have already obtained the control over registers `rsi` and `rdx` pertaining to the second and third parameters of `copy_from_user` respectively. Leveraging the control over registers, we manipulate the values in these registers. More specifically, we set `rsi` and `rdx` to $p1 + \text{PAGE\_SIZE} - n$ and $n+1$ respectively, with n representing the length of payload actual copied. When the function attempts to copy the last byte, it failes to access the content at $p2$ and triggers a page fault because the page $p2$ is not mapped into the memory. Eventually `copy_from_user` returns a positive number 1 because one byte is not successfully copied.

6.2 Bypassing Stack Canary

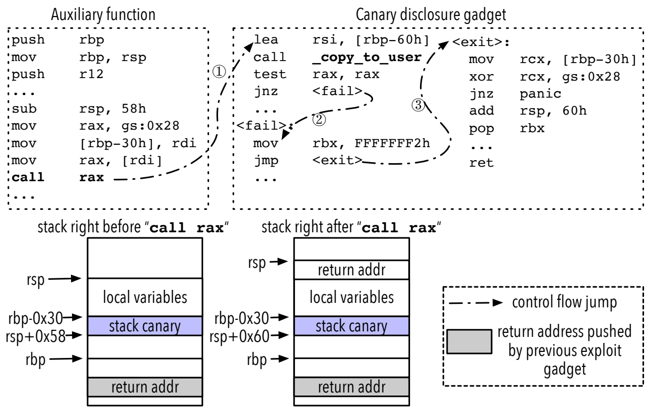
As is mentioned earlier, to prepare a working payload for stack smash, an attacker has to know the value of kernel stack canary which remains secret in our threat model. We consider the presence of a strong kernel stack canary setting where stack canary is enabled for all functions containing a local variable.

We will first introduce two kinds of prevalent gadgets, then we discuss how to pair them to dump kernel stack memory.

6.2.1 Exposing Stack-disclosure Gadget and Auxiliary Function

To leak stack canary, an intuitive way is to construct an info leak of its value to user-space through an official data channel such that `SMAP` is not violated. In the following we introduce stack-disclosure gadget which is twin gadget of stack-smash gadget as well as auxiliary function prologue gadget.

Stack-disclosure gadget. Function `copy_to_user()` is widely used in the Linux kernel codebase to copy kernel memory into user-space. In Linux kernel 4.15, our statistic indicates this function has been invoked at 594 sites. Of all these invocations, 82% are used for copying data from kernel



Auxiliary function:

```
push    rbp
mov     rbp, rsp
push    r12
...
sub     rsp, 58h
mov     rax, gs:0x28
mov     [rbp-30h], rdi
mov     rax, [rdi]
call    rax
...
```

Canary disclosure gadget:

```
  lea     rsi, [rbp-60h]      <exit>:
  call    _copy_to_user         mov   rcx, [rbp-30h]
  test    rax, rax              xor   rcx, gs:0x28
  jnz     <fail>                jnz   panic
  ...                           add   rsp, 60h
<fail>:                         pop   rbx
  mov     rbx, FFFFFFF2h        ...
  jmp     <exit>                ret
  ...
```

Figure 6: An example of canary disclosure gadget and its corresponding auxiliary gadget.

stack to user-space. Since this naturally establish a channel to migrate data from kernel stack to user-space, we could exploit the characteristic of this kernel function to disclose the canary on kernel stack.

Auxiliary function. To successfully return from a stack-disclosure gadget and continue exploitation, we use auxiliary function to create a similar stack frame as the stack-disclosure gadget and transfer the control-flow to stack-disclosure gadget with an indirect call after the function prologue.

Auxiliary function should have stack canary protection and contain a controllable indirect call after its own function prologue which establishes a stack frame. Its layout of stack frame could be paired with a stack-disclosure gadget to form a "complete" stack frame and pass the stack canary check by putting a valid stack canary on the stack.

6.2.2 Disclosing Canary on Kernel Stack

By diverting the control-flow to a call site of `copy_to_user()`, we are closer to successfully disclose stack contents by satisfying the following four requirements. ❶, the registers should be set properly as parameters for `copy_to_user`, ❷, the kernel should not panic during the path caller function returns, ❸, the caller function of `copy_to_user` checks stack canary before return, ❹, the return address on stack must be set properly to continue the rest of the exploitation.

To tackle the first requirement, we leverage *blooming gadget* described in Section 6.3. For the second requirement, we could *trigger a page fault* and take a *short return path* similar to the technique described in Section 6.1.2. For the last two requirements, we pair stack-disclosure gadget with auxiliary function to generate a valid stack frame.

The key insight behind using auxiliary function to pair with stack-disclosure gadget is reusing the canary generated by the prologue of auxiliary function. A pair of them should have the same number of saved registers, the same canary location and stack size of 8 bytes difference.

```
static void aliasing_gtt_unbind_vma(struct
     i915_vma *vma) {
   ...
   vma->vm->clear_range(vma->vm, vma->node.start,
        vma->size);
   ...
}
```

Table 3: The kernel code fragment (a blooming gadget) that could enhance the control over multiple general registers.

To elucidate the rationale behind the pairing, we take for example the routine of canary disclosure in Figure 6. We re-direct a `CFHP` to auxiliary function, After the prologue of auxiliary function which saves registers and establishes a stack frame, the target of indirect call `call rax` is set to the stack disclosure gadget in ①. Then content of current stack frame is copied to user-space by `copy_to_user`, a page fault is triggered to force non-zero return value of `copy_to_user`, as result *short return path* is taken in ②. Before the function returns, stack canary sanity check is performed ③, because the auxiliary function put a valid stack canary in current stack frame, the canary check is successfully passed and return to the caller of auxiliary function.

6.3 Putting them together: "Single-shot" Exploitation

It remains challenging to use an ill-suited `CFHP` to first disclose stack canary and then smashing kernel stack. The reason behind is a `CFHP` in practice may have limitations in the following two aspects. First, difficulty in combining aforementioned two building blocks with a single `CFHP`, second, lack of register control. "Single-shot" exploitation uses a *blooming gadget* to amplify control over other registers and a *bridging gadget* to combine the two actions sequentially.

6.3.1 Augmenting CFHP with Blooming Gadget

To enhance a `CFHP` with the ability to control more registers, we introduce a family of *blooming gadgets*. The use of blooming gadget is inspired by `COOP` [59] which exploits a series of type confusions C++ program. Although Linux is written in C, its code heavily exhibits feature of object-oriented programming. The "self" object is usually passed as the first argument of function through `rdi`. And oftentimes the function contains‘‘ indirect call using function pointer that resides in the object passed as parameter. We could let the `CFHP` to land at these functions to abuse type confusion.

We illustrate one such blooming gadget in Table 3. Kernel function `aliasing_gtt_unbind_vma()` contains an indirect call with three parameters calculated by dereferencing the function's first parameter `*vma`. Assume we have a `CFHP` with

```
1  void regcache_mark_dirty(struct regmap *map) {
2      map->lock(map->lock_arg); // the 1st
           control-flow hijack
3      map->cache_dirty = true;
4      map->no_sync_defaults = true;
5      map->unlock(map->lock_arg); // the 2nd
           control-flow hijack
6  }
```

Table 4: The source code of a bridging gadget – the kernel code fragment that could spawn multiple CFHPs.

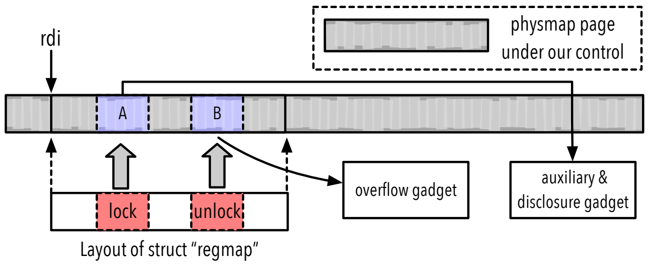

rdi
physmap page under our control
A
B
lock
unlock
overflow gadget
auxiliary & disclosure gadget
Layout of struct "regmap"

Figure 7: Memory layout after using physmap spray [39] to allocate physmap pages with data under our control.

control over rdi, we can get an augmented CFHP which controls rdi, rsi, and rdx at the same time at line 3 in Table 3.

Note a blooming gadget works only if rdi is controllable at beginning. We found this requirement is easy to fullfil in practice. Our insight is that a CFHP usually has one register potentially controllable - either the register is fully controllable or the register points to a heap area under control. Such primitive can be turned into a CFHP with rdi control easily through a single gadget which ends with an indirect call. A worst case happens where a CFHP implies none of potentially controllable registers. Fortunately, we are still able to leverage uninitialized or controllable data on kernel stack as well as a common ROP gadget. For example, we could use the gadget `add rsp, 0x68; pop rdi; ret` to gain control over rdi if rsp+0x68 and rsp+0x70 is under our control.

6.3.2 Spawning Multiple CFHPs with Bridging Gadget

As is demonstrated in the motivating example in Section 4, an exploitation practice depending on re-triggerable CFHP is not reliable because of exploit path pitfalls. We use bridging gadget - a family of kernel functions with multiple controllable indirect calls - to spawns two CFHPs and combine canary leak and stack smash into a single shot.

For example, function regcache_mark_dirty shown in Table 4 is such a bridging gadget which contains two indirect calls, map->lock in line 2 and map->unlock in line 5.

As we can observe from the kernel gadget shown in Table 4, the function pointers tied to these two indirect calls are enclosed in a data object referred by the first argument

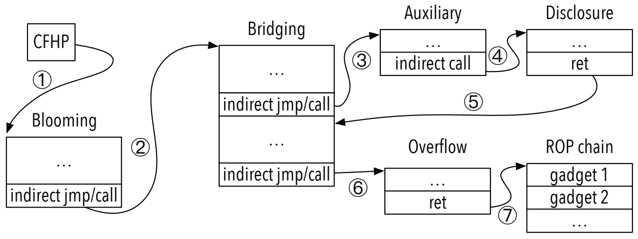


Figure 8: An illustration of how KEPLER stitches various kernel gadgets for ultimate exploitation.

of the function regcache_mark_dirty(). Recall that the first argument of a function is specified by the general register rdi, and we can usually obtain the control over that register using technique described in Section 6.3.1. As a result, in order to obtain control over both function pointers, we could first employ ret2dir [39] to allocate physmap pages and carefully crafted a data object accordingly. Then, we could refer the register rdi to a proper spot and set rip to the entry site of the gadget shown in Table 4. As is shown in Figure 7, assume the data carefully crafted in the spots of A and B represent the address of the auxiliary gadget together with a disclosure gadget responsible for leaking stack canary as well as the entry address of the gadget pertaining to stack smashing respectively. Then, by executing the bridging gadget shown in Table 4, we could first leak canary using the first indirect call. After the return of the call to copy_to_user(), there is no operations between the consecutive indirect calls that impose additional constraint to the second function pointer. Therefore, we could perform the stack smashing using the second function pointer without involving unexpected termination.

7 Implementation

Using IDA Pro SDK [20] and angr [64], we implemented KEPLER with about 8,000 lines of Python code. KEPLER is an automated tool that tracks down the aforementioned exploitation gadgets and chains them for exploitability assessment. Figure 8 depicts how these gadgets are concatenated. While previous sections have discussed the basic building blocks to perform an "single-shot" exploitation, the exploitation chain could not be determined once and for all with static analysis because uniqueness of each CFHP and different gadget combinations bring about the variation of the exploitation context. For example, the consecutive exploitation gadgets might no longer obtain the control over related registers with a different initial CFHP.

To address this problem, we developed our tool to assess each of the gadget chains potentially useful for kernel exploitation. More specifically, we follow the guidance of the exploitation chain construction shown in Figure 8, and design our tool to perform a depth-first exhaustive search which explores all the possible combinations of exploitation gadgets. When performing the depth-first search: starting from the rip hijack site, KEPLER symbolically executes the gadget chain that the search algorithm explores. To determine whether a gadget chain is useful for exploitation, our tool checks the memory access and deems a gadget chain useless if that exploitation chain attempts to access the user space or an unmapped kernel memory region. In addition, KEPLER examines the control over the registers at two critical sites – one at the entry of the disclosure gadget and the other at the entry of the overflow gadget. We implemented KEPLER to deem a gadget chain useless and terminate symbolic execution earlier if it has no control over the registers rdi and rdx at the first checking site or has no control over rsi and rdx at the second checking site. The reason behind this implementation is that, after executing bridging and auxiliary gadgets, we might lose the control over the registers needed for disclosure and overflow gadgets. With the check right before symbolically executing the two gadgets, we can quickly determine the usefulness of the gadget chain in exploitation, terminate unnecessary symbolic execution and thus save the computation resources.

In the process of the assessment of the exploitation chain, KEPLER symbolically executes each exploitation gadget. For some of them, they might carry a large number of basic blocks and even infinite loops. This could significantly influence the efficiency our tool and even incur the state explosion problem. To avoid these issues, for each path in an exploitation gadget, we set KEPLER to explore at most 20 basic blocks. In addition, we developed KEPLER to concretize each symbolic address. To be more specific, we set up a kernel page under our control in the physmap region and then concretize each symbolic address with a non-overlapping address of that memory page. In this work, we implemented this concretization mechanism by simply extending ControlledData – one of the symbolic address concretization strategies of angr.

For each gadget chain that passes the assessment, KEPLER further performs constraint solving to generate payload accordingly. Technically, this can be easily done by using angr. However, the Z3 solver used in angr consumes memory exhaustively and generally does not release the memory used for constraint solving even after the completion of computation. To address this problem, KEPLER partitions symbolic execution and constraint solving into two different processes. In this way, KEPLER could terminate the memory-intensive process every time the constraint solving is completed and thus free the memory for consecutive computation.

As is described in Section 6.1, the payload smashed to the stack contains the stack canary disclosed as well as a sequence of addresses indicating an ROP chain that performs actual exploitation. With respect to the stack canary, we employ a separated thread in the user-space to rapidly retrieve the canary – whenever it is disclosed to the user-space – and then make it ready for stack smashing. Regarding the ROP chain used in this work, we simply choose the ROP payload commonly used for privilege escalation. In Appendix, we specify the ROP payload used in this work. It invokes kernel functions commit_creds() and prepare_kernel_cred() to obtain the root privilege. Note that the construction of an ROP payload is out of scope of this paper. There are many commonly-adopted ROP payloads, which can be naturally hooked with our new kernel exploitation technique.

8 Case Study and Evaluation

In this section, we demonstrate our new exploit technique and evaluate our automated tool KEPLER using real-world kernel vulnerabilities and some recently-released CTF challenges. To be specific, we compare KEPLER with various kernel exploitation techniques to show it is an effective exploit technique, we also compare KEPLER with automatic exploit generation systems to highlight its power in evaluating exploitability with a CFHP. In addition, we show the efficacy and efficiency of KEPLER in facilitating exploitation chain construction.

8.1 Setup

We first randomly selected 3 recently released CTF challenges as well as 16 real-world kernel vulnerabilities archived between 2016 and 2017. Then, we successfully assembled these vulnerabilities in a mainline Linux kernel 4.15.0 by inserting them into the kernel code or reverting their patch accordingly. In this work, we evaluate our tool KEPLER by using this single Linux kernel, and demonstrate the effectiveness of "single-shot" exploitation by launching exploitations against the inserted vulnerabilities.

As is summarized in Table 5, the vulnerabilities inserted cover various types such as Use-After-Free and Out-Of-Bound (OOB) read/write etc. It should be noted that the CVEs selected are a little bit unbalanced – with more in 2017 and less in 2016. On the one hand, this is because there are more than $2\times$ of kernel vulnerabilities reported in 2017 than those in 2016 [1]. On the other hand, this is because some components in Linux kernel experience significant overhaul since 2016 and we have difficulty of re-enabling the corresponding vulnerabilities in a new kernel image.

In order to run and evaluate KEPLER, we also assembled and configured a testbed which has a 32-core Intel(R) Xeon(R) Platinum 8124M CPU and 256GB of memory. For each vulnerability, we then used this testbed to run 28 concurrent workers which symbolically explore the kernel code space and track down useful exploitation chains in parallel.

8.2 Effectiveness of "single-shot" exploitation

By searching the Internet, we gathered 10 exploits pertaining to the vulnerabilities inserted. As is shown in Table 5, these

ID	Vulnerability type	Public exploit	Q	FUZE	KEPLER	G1	G2	G3	G4	First chain (min)	Total time (hour)	Total # of exploitation chains
CVE-2017-16995	OOB readwrite	✓†	✗	✗	✓	41	114	27	201	45	37	29788
CVE-2017-15649	use-after-free	✓	✗	✓	✓	29	79	25	280	16	28	60207
CVE-2017-10661	use-after-free	✗	✗	✗	✓	28	78	30	301	17	25	49070
CVE-2017-8890	use-after-free	✗	✗	✗	✓	21	88	23	304	17	18	50471
CVE-2017-8824	use-after-free	✓	✗	✓	✓	63	101	35	306	50	70	164898
CVE-2017-7308	heap overflow	✓	✗	✗	✓	31	91	30	241	14	47	110176
CVE-2017-7184	heap overflow	✓	✗	✗	✓	31	95	31	254	24	37	93752
CVE-2017-6074	double-free	✓	✗	✗	✓	18	79	31	308	16	15	31436
CVE-2017-5123	OOB write	✓†	✗	✗	✓	40	86	27	311	14	39	113466
CVE-2017-2636	double-free	✗	✗	✗	✓	18	89	29	289	29	19	26372
CVE-2016-10150	use-after-free	✗	✗	✗	✓	34	84	25	293	52	34	88499
CVE-2016-8655	use-after-free	✓†	✗	✓†	✓	18	109	32	260	15	17	47413
CVE-2016-6187	heap overflow	✗	✗	✗	✓	22	85	32	301	17	21	51954
CVE-2016-4557	use-after-free	✗	✗	✗	✓	21	80	21	295	16	37	40889
CVE-2017-17053	use-after-free	✗	✗	✗	✗	-	-	-	-	-	-	-
CVE-2016-9793	integer overflow	✗	✗	✗	✗	-	-	-	-	-	-	-
TCTF-credjar	use-after-free	✓†	✗	✗	✓	35	89	25	292	25	14	82913
0CTF-knote	uninitialized use	✗	✗	✗	✓	21	89	33	318	17	36	40923
CSAW-stringIPC	OOB read&write	✓†	✗	✗	✓	35	88	25	289	17	33	84414

Table 5: The comparison of exploitability as well as performance of KEPLER. G1, G2, G3 and G4 represent the blooming gadget, bridging gadget, auxiliary and disclosure gadget pair, and stack-smash gadget. The "first chain" column indicates the time spent on pinpointing the first exploitation chain. The "total time" column specifies the total amount of time spent on finding all useful exploitation chains. † symbol represents the cases where the exploits could only bypass major mitigations (*e.g.,* SMAP and SMEP) and fail to bypass others under our threat model. ✓and ✗ symbols indicate the existence and non-existence of a working exploit.

publicly available exploits perform exploitation through various approaches and therefore demonstrate different capability in bypassing kernel mitigations. Among these exploits, we found there are only 5 of them demonstrating the ability to perform exploitation under our aforementioned threat model. In comparison with the working exploits generated by KEPLER, publicly available exploits demonstrate much weaker exploitability (with 5 vs 17 cases). To some extent, this implies existing exploitation approaches highly rely upon the quality of the target vulnerability and corresponding CFHP, whereas our approach KEPLER could utilize prevalent kernel function and gadgets to explore exploitable machine states and thus escalate the exploitability for a CFHP. However, previous exploit technique Q [60] could not generate working exploit because Q rely on a stack pivoting gadget while its gadget discovery phase return none of working pivoting gadget[4]. FUZE could only generate exploit for 3 cases because it evaluate exploitability of a CFHP simply with two straight forward exploit technique: `pivot-2-usr` and "`cr4`-flipping". The former does not bypass SMAP and the latter only works when at least two CHFPs is available.

Even for the vulnerabilities against which both public exploits and ours demonstrate the same capabilities in bypassing mitigations, we argue that our approach still exhibits stronger exploitability. This is because the public exploits circumvent mitigations by manipulating control registers with two CFHPs, as is discussed in Section 2.3, this practice can be easily restricted by virtualization extension. For the two vulnerabilities CVE-2017-17053 and CVE-2016-9793 for which our approach fails to derive working exploits, we manually examine their execution traces leading to the kernel panic. We find that the failure results from the following fact. In order to take the control over `rip` prior to exploitation, both of these vulnerabilities require an exploit to access the data in the user space. This violates the protection of SMAP. KEPLER restricts any operations that violate our threat model and output a failure if none of the exploitation chains could avoid such violation.

8.3 Effectiveness and Efficiency of Our Tool

Our experiment utilizes KEPLER to explore the aforementioned kernel image with the vulnerabilities inserted. In this process, we exhaustively search gadget chains useful for exploitation and mitigation circumvention. In Table 5, we show the total number of useful exploitation chains identified as well as the total amount of time spent on finding these gadget chains. As we can observe, KEPLER could automatically pinpoint tens of thousands of unique kernel gadget chains to perform exploitation without triggering kernel protections. Since we implement KEPLER to perform gadget chain exploration in parallel, we also discover that these gadget chains could typically be identified within 50 hours. These observations together imply that KEPLER could diversify the ways of performing kernel exploitation in an efficient fashion. Given that some commercial security products pinpoint kernel exploita-

[4]The result related to Q in our evaluation is based on inference of its design instead of running its tool because we were not able to get the source code of Q.

tion by using the patterns of exploits, the ability to diversify exploitation has the potential to assist an adversary to bypass the detection of commercial security products.

From Table 5, we also observe that, for different vulnerabilities, KEPLER generates different number of gadget chains useful for exploitation. This can be attributed to the following fact. In the process of gadget chain identification and assessment, KEPLER starts gadget assessment from different machine states and contexts. For some vulnerabilities, the machine states and contexts do not provide us with sufficient control over some registers and memory regions. Under this circumstance, the availability of useful kernel gadgets would vary and thus influence the total number of generated exploits.

In Table 5, we also depict the time spent on finding the first kernel gadget chain useful for exploitation. As we can observe, KEPLER could quickly output an useful exploitation chain in less than about 50 wall-clock minutes (and the corresponding CPU-core time is roughly 1400 minutes given the prototype system uses 28 concurrent workers). This implies KEPLER has the potential to be used as a tool to quickly derive a working exploit without too many human efforts. Last but not least, Table 5 also shows the total number kernel gadgets in different categories. As we can observe, there are typically tens of gadgets in each categories. This means that one cannot simply block our exploitation approach by eliminating a small number of kernel gadgets. In Section 9, we will further discuss the defense of our exploitation approach.

9 Discussion and Future Research

In this section, we discuss some plausible defence mechanisms against our "single-shot" exploitation chain. Also, we elaborate why they are not effective nor suitable for preventing the proposed attack. Following our discussion and analysis, we then provide some suggestions for the future research.

Plausible Defense Mechanisms. To defend against the exploit chain mentioned above, one straightforward reaction is to eliminate the gadgets that must be used in kernel exploitation. However, as we have already demonstrated and discussed in Section 8, the tool we develop could enrich the choices of the gadgets needed for exploitation. This means that, following this potential solution, Linux developers would inevitably introduce significant amount of kernel code changes and it is difficult to guarantee these changes would not bring about negative influence upon Linux kernel execution. Other security mitigation could also be used as potential defense mechanisms. For example, there have already been a rich collection of research works on control and data flow integrity protection (e.g. [2] [78] [79] [25] [16] [26]). In addition, randomizing stack canary per-function call [73] could idealy prevent our exploit technique because it discourage the effort to fake stack frame and leak stack canary with `copy_to_user`. Integrating and enabling them in Linux kernel, they could easily fail the attack mentioned above. Unfortunately, these techniques usually incur unacceptable overhead (e.g., [16] has an average overhead of 13%) or sometimes rely upon hardware features to reduce their overhead (e.g., [25]). As a result, they are barely used as a practical, general defense solution in popular release version of Linux kernel.

Possible Future Research. Looking ahead, we suggest the future research could be conducted from two aspects. From the perspective of automatic exploit primitive evaluation, we believe there is an emerging need to invent technique to systematically evaluate various exploit primitives, expecially for those weak exploit primitives. In practice, theory and techniques should be proposed to facilitate deriving better exploit primitive with a initially weak exploit primitive. From the perspective of defense, on one hand, we believe there remains the need to design lightweight control-flow enforcements for Linux kernel. On the other hand, instead of manually overhauling kernel code, one could augment GCC with the ability to eliminate the exploitation gadgets at compilation time.

10 Conclusion

We show it is generally challenging to generate exploits with a control-flow hijacking primitive in the Linux kernel under a realistic threat model, while there are a lot of research efforts in identifying exploit primitives and facilitating exploit generation with various exploit primitives. We propose KEPLER, a framework to facilitate evaluation of control-flow hijacking primitives which leverages a novel "single-shot" exploitation to convert a control-flow hijacking primitive into a classic stack overflow and thus bootstrap traditional code-reuse attack against modern Linux kernel. In comparison with previous automatic exploit generation and exploit hardening techniques, we show that KEPLER outperforms other exploit techniques and is able to generate thousands of exploit chains for a control-flow hijacking primitive in Linux kernel despite the challenges of widely-deployed security mitigations, exploit path pitfalls and ill-suited exploit primitives. Following the experimental results, we safely conclude that KEPLER can significantly facilitate evaluating control-flow hijacking primitive in the Linux kernel.

11 Availability

We release the source code of KEPLER, a kernel embeded with vulnerabilities and generated gadget chains for research and education purposes [76].

12 Acknowledgements

We would like to thank our shepherd Stephen McCamant and anonymous reviewers for their help and comments. The IIE authors were partially supported by the Stategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), the National Key Research and Development Program of China (2016YFB0801004, 2016QY071405, 2018YFB0803602, 2016QY06X1204), the Key Foundation of Beijing Committee of Science and Technology (Z181100002718002), the Key Laboratory of Network Assessment Technology of Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology. The PSU authors were partially supported by IST seed grant. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements.

References

[1] Linux kernel vulnerability statistics, 2018. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12nd ACM conference on Computer and communications security (CCS)*, 2005.

[3] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57, 2014.

[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[5] T. Bao, R. Wang, Y. Shoshitaishvili, and D. 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.

[6] S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. {*USENIX; login:*}, 2011.

[7] D. Brumley, P. Poosankam, D. X. Song, and J. 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.

[8] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of IEEE Symposium on Security and Privacy (SP), 2012*, 2012.

[10] K. Cook. x86/mm: Always enable config_debug_rodata and remove the kconfig option, 2016. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ccaf77cf05915f51231d158abfd5448aedde758.

[11] J. Corbet. Supervisor mode access prevention, 2012. https://lwn.net/Articles/517475/.

[12] J. Corbet. Post-init read-only memory, 2015. https://lwn.net/Articles/666550/.

[13] J. Corbet. A page-table isolation update, 2018. https://lwn.net/Articles/752621/.

[14] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[15] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[16] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[17] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[18] david942j. The one-gadget in glibc, 2018. https://david942j.blogspot.com/2017/02/project-one-gadget-in-glibc.html.

[19] dong-hoon you. New reliable android kernel root exploitation techniques, 2016. http://powerofcommunity.net/poc2016/x82.pdf.

[20] C. Eagle. *The IDA Pro Book (Second edition)*. no starch press, 2011.

[21] J. Edge. Extending the use of ro and nx, 2011. https://lwn.net/Articles/422487/.

[22] J. Edge. "strong" stack protection for gcc, 2014. https://lwn.net/Articles/584225/.

[23] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[24] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden. Pshape: Automatically combining gadgets for arbitrary method execution. In *International Workshop on Security and Trust Management*, 2016.

[25] X. Ge, W. Cui, and T. Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGOPS Operating Systems Review*, 51(2), 2017.

[26] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings of 2016 IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[27] J. Gionta, W. Enck, and P. Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS)*, 2016.

[28] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel

ASLR. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[29] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[30] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[31] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24nd USENIX Security Symposium (USENIX Security)*, 2015.

[32] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[33] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*, 2009.

[34] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[35] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 2018.

[36] itsZN. Bypassing smep using vdso overwrites, 2015. https://itszn.com/blog/?p=21.

[37] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[38] M. Jurczyk and G. Coldwind. SMEP: What is it, and how to beat it on windows, 2011. http://j00ru.vexillium.org/?p=783.

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

[40] S. Knox. Real-time kernel protection(rkp), 2016. https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp.

[41] A. Konovalov. Exploiting the linux kernel via packet sockets, 2017. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[42] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[43] G. Kroah-Hartman. Introduce static_usermodehelper to mediate call_usermodehelper, 2017. https://patchwork.kernel.org/patch/9519063/.

[44] Lexfo. Cve-2017-11176: A step-by-step linux kernel exploitation, 2018. https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html#stack-pivoting.

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Procceedings of 27th USENIX Security Symposium (USENIX Security)*, 2018.

[46] K. Lu, M. Walter, D. Pfaff, and S. 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.

[47] M. Miller. Modeling the exploitation and mitigation of memory safety vulnerabilities. In *Breakpoint*, 2012.

[48] R. Mothe and R. R. Branco. Dptrace: Dual purpose trace for exploitability analysis of program crashes. In *Black Hat USA Briefings*, 2016.

[49] J. Nakajima and S. Grandhi. Kernel protection using hardware based virtualization. In *The Linux Foundation events*, 2017.

[50] National Vulnerability Database. Cve-2017-8890 detail, 2017. https://nvd.nist.gov/vuln/detail/CVE-2017-8890.

[51] M. Oh. Detecting and mitigating elevation-of-privilege exploit for cve-2017-0005, 2017. https://cloudblogs.microsoft.com/microsoftsecure/2017/03/27/detecting-and-mitigating-elevation-of-privilege-exploit-for-cve-2017-0005/.

[52] S. Pailoor, A. Aday, and S. Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[53] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis. Kernel protection against just-in-time code reuse. *ACM Transactions on Privacy and Security (TOPS)*, 22(1), 2019.

[54] A. Prakash and H. Yin. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 111–120, 2015.

[55] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[56] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[57] J. Salwan. Ropgadget, 2012. https://github.com/JonathanSalwan/ROPgadget.

[58] S. Schumilo, C. Aschermann, and R. Gawlik. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2017.

[59] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications.

In *In Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, 2011.

[61] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges, 2015. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[62] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, et al. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy*, 16(2):12–22, 2018.

[63] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[64] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. 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.

[65] K. A. Shutemov. pagemap: do not leak physical addresses to non-privileged userspace, 2015. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[66] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34 IEEE Symposium on Security and Privacy (S&P)*, 2013.

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

[68] spender. wait for kaslr to be effective, 2017. https://grsecurity.net/~spender/exploits/wait_for_kaslr_to_be_effective.c.

[69] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[70] P. Team. Rap: Rip rop, 2015. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[71] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

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

```
int i=0;
unsigned long *p=(unsigned long*)PAYLOAD_START;
p[i++]=0; // padding
p[i++]=0; // canary location
p[i++]=0; // padding for saved registers
...
// priviledge escalation
p[i++]=POPRDI; // pop rdi ; ret
p[i++]=0;
p[i++]=PREPARE_KERNEL_CREDS;
p[i++]=POPRDXRET; // pop rdx ; ret
p[i++]=COMMIT_CREDS;
p[i++]=MOV_RDI_RAX_JMP_RDX; // mov rdi, rax ;
    jmp rdx
// sleep for 60 minutes
p[i++]=POPRDI; // pop rdi ; ret
p[i++]=1000 * 60 * 60;
p[i++]=MSLEEP;
```

Table 6: The kernel ROP payload that performs privilege escalation.

[73] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao. To detect stack buffer overflow with polymorphic canaries. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 243–254. IEEE, 2018.

[74] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

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

[76] ww9210. kepler-cfhp, 2018. https://github.com/ww9210/kepler-cfhp.

[77] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. 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 (CCS)*, 2015.

[78] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[79] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

Appendix

As is discussed in Section 6, we utilize a series of kernel gadgets to bypass kernel mitigations. After that, we redirect the control flow of the Linux kernel to a universal ROP payload. By using that payload,

```
// copy more payloads to the kernel stack
// prepare auguments for copy_from_user()
p[i++]=POPRAX; // pop rax ; ret
p[i++]=POPRSI; // pop rsi ; ret
p[i++]=0xffffffff81254a99; // mov rdi, rsp ;
     call rax
p[i++]=POPRAX;
p[i++]=0x1000;
p[i++]=0xffffffff81a04201; // sub rdi, rax ;
     mov rax, rdi ; ret
p[i++]=POPRSI;
p[i++]=STAGE_TWO_ROP_PAYLOAD;
p[i++]=POPRDX; // pop rds ; ret
p[i++]=0x1040;
p[i++]=COPY_FROM_USER; // copy_from_user()

// substract rsp to the first gadget
p[i++]=POPRAX;
p[i++]=0x1040;
p[i++]=0xffffffff81a04201; // sub rdi, rax ;
     mov rax, rdi ; ret
p[i++]=POPR12; // pop r12 ; ret
p[i++]=0xffffffff810001cc; // ret
p[i++]=0xffffffff81c01688; // mov rsp, rax ;
     push r12 ; ret
```

Table 7: The kernel ROP payload that copies an ROP payload to the current stack frame and then subtracts the stack pointer to execute the ROP payload.

we demonstrate the exploitability of a kernel vulnerability. In Table 6, we show an ROP payload used in this work. As is specified, it first performs privilege escalation. Then, it sets the Linux kernel to fall into asleep for a long time by using the kernel function `msleep()`.

Considering Linux kernel might perform inline permission checks and we need to execute an ROP payload with an arbitrary length, we further utilize the ROP payload like the one shown in Table 7 to address this payload length issue.