Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers

Haehyun Cho[1], Jinbum Park[2], Joonwon Kang[2], Tiffany Bao[1],
Ruoyu Wang[1], Yan Shoshitaishvili[1], Adam Doupé[1], Gail-Joon Ahn[1,2]

[1]*Arizona State University* [2]*Samsung Research*

{haehyun, tbao, fishw, yans, doupe, gahn}@asu.edu, {jinb.park, joonwon.kang}@samsung.com

Abstract

Information leaks are the most prevalent type of vulnerabilities among all known vulnerabilities in Linux kernel. Many of them are caused by the use of uninitialized variables or data structures. It is generally believed that the majority of information leaks in Linux kernel are low-risk and do not have severe impact due to the difficulty (or even the impossibility) of exploitation. As a result, developers and security analysts do not pay enough attention to mitigating these vulnerabilities. Consequently, these vulnerabilities are usually assigned low CVSS scores or without any CVEs assigned. Moreover, many patches that address uninitialized data use bugs in Linux kernel are not accepted, leaving billions of Linux systems vulnerable.

Nonetheless, information leak vulnerabilities in Linux kernel are not as low-risk as people believe. In this paper, we present a generic approach that converts stack-based information leaks in Linux kernel into kernel-pointer leaks, which can be used to defeat modern security defenses such as KASLR. Taking an exploit that triggers an information leak in Linux kernel, our approach automatically converts it into a highly impactful exploit that leaks pointers to either kernel functions or the kernel stack. We evaluate our approach on four known CVEs and one security patch in Linux kernel and demonstrate its effectiveness. Our findings provide solid evidence for Linux kernel developers and security analysts to treat information leaks in Linux kernel more seriously.

1 Introduction

For performance concerns, unsafe programming languages, such as C and C++, are still prevalently used in the implementation of operating system (OS) kernels and embedded systems. While these unsafe languages may allocate memory on stack or in the heap for variables, these variables may not be initialized before being used. When a variable is used without proper initialization (which can be caused by either a programming mistake or padding bytes in a struct inserted by compilers [22]), the memory values that were present at the same location of the variable before it was allocated—called *stale values*—will be read and used. When these stale values are copied from the kernel space to the user space, user-space programs will be able to access them, which causes an information-leak vulnerability if the information contained in the stale values is important.

The use of stale values in Linux kernels can lead to severe security problems, which have been studied in the past [10, 16, 21]. Moreover, these stale values can pose severe security threats without being directly used in the kernel. For example, modern kernel security defenses, such as Kernel Address Space Layout Randomization (KASLR), depend on keeping kernel addresses secret from user-space programs. When attackers get lucky and recover kernel pointer values through leaked information (stale values) from the kernel space, they can defeat KASLR [15, 25]. Likewise, attackers may leak cryptographic keys that are stored in the kernel space.

Unfortunately, in Linux kernel, information leaks that are caused by uninitialized data are common. A study shows that information leak vulnerabilities that are caused by the use of uninitialized data are the most prevalent type of vulnerabilities among the four major types of vulnerabilities in Linux kernel [8]. Within the past two years, KernelMemorySanitizer (KMSAN) discovered over 100 uninitialized data use bugs in Linux kernel through fuzzing [5]. Worse, due to the difficulty (or the impossibility) of exploiting the majority of information leak vulnerabilities or using them in high-risk exploits (such as remote code execution or local privilege escalation), these vulnerabilities are commonly believed to be of low risks. As a result, many uninitialized data uses do not get sufficient attention from developers or security researchers, are not assigned any CVE entries[1], and in some cases their corresponding patches are not merged into Linux kernel for a long time [16].

[1] Here is an example of a security patch that fixes a stack-based information leak vulnerability: https://github.com/torvalds/linux/commit/7c8a61d9ee. No CVE was ever assigned for the vulnerability.

	Total	Stack-based	Heap-based	# of exploits
# of CVEs	87	76 (87%)	11 (13%)	0

Table 1: The number of information leak CVEs that are related to uses of uninitialized data between 2010 and 2019. The majority of these CVEs are stack-based information leaks. There are no publicly available exploits for these CVEs. Only one out of these 87 CVEs warns about possible leaks of kernel pointers and potential KASLR bypasses.

Table 1 shows the statistics of 87 Linux kernel CVEs that are related to uninitialized data uses and are reported between 2010 and 2019 [6]. The majority of these CVEs are stack-based information leaks. Evaluating the severity of these CVEs is extremely difficult since no public exploit is available for any of them. Even if a public exploit is available, using these vulnerabilities to leak key information usually requires manual and complicated manipulation of the kernel layout, which is costly and time-consuming. Therefore, all but one CVE (CVE-2017-1000410) mentions anything about the potential of leaking kernel pointers and bypassing KASLR, which leaves an impression to the general public that these vulnerabilities are of low security impact.

The situation about information leaks in Linux kernel is extremely concerning. In this paper, we demonstrate the *actual* exploitability and severity of information leak bugs in Linux kernels by proposing a generic and automated approach that converts stack-based information leaks in Linux kernels into vulnerabilities that leak kernel pointer values. Specifically, we focus on leaking pointer values that point to kernel functions or the kernel stack. These leaked kernel pointer values can be used to bypass kernel defenses such as KASLR, which is an essential step in modern Linux kernel exploits [13].

Our proposed approach takes as input an exploit that triggers a stack-based information leak bug, analyzes the exploit to identify locations where stale values are coming from, and reasons about an attack vector that places kernel pointer values at these locations. It is worth mentioning that our approach supports leaking kernel pointers when the size of the leaked stale value is less than a full 64-bit pointer (8 bytes). We evaluate our approach on five real-world Linux kernel vulnerabilities (including four CVEs and one bug that was reported by KMSAN) and demonstrate its generality and effectiveness. The existing Common Vulnerability Scoring System (CVSS) scores of three of the above CVEs are 2.1 (on a scale of 0 to 10, higher is more severe), which imply that "specialized access conditions or extenuating circumstances do not exist, even though there is considerable informational disclosure" [2–4]. Our findings can be used to assist CVSS in correcting the scoring and assessment of information leak vulnerabilities in Linux kernels, and raise awareness in the security community of these vulnerabilities.

Contributions. This paper makes the following contributions:

- We disclose the actual severity of information leak vulnerabilities in Linux kernel. These vulnerabilities are easier to exploit and are more severe than what is generally believed.
- We identify common challenges in exploiting information-leak vulnerabilities. We then propose a generic and automated approach that converts a stack-based information leak vulnerability in Linux kernel to an exploit that leaks kernel pointer values.
- We implement our approach and evaluate it on five real-world vulnerabilities (four CVEs and one fixed bug in the upstream Linux kernel). The evaluation results show that our proposed approach is effective.

In the spirit of open science, we have released the source code of our tool and the exploits that we developed as part of our research. The repository is at `https://github.com/sefcom/leak-kptr`.

2 Background

In this section, we provide a technical overview of stack-based information leaks in Linux kernel, and how leaked kernel pointer values can be used in more severe types of kernel exploits.

2.1 Information Leaks from the Kernel Stack

Each thread in the Linux has a kernel stack, which is a memory area that is allocated in kernel space. Depending on the specific Linux version and configuration, the sizes of the kernel stack differ. To maximize the locality, the kernel stacks are usually small in size (8KB or 16KB on x86-64). Therefore, the memory space for the kernel stack is very frequently reused between different kernel stack frames. Lu, *et al.* showed that 90% syscalls only use less than 1,260 bytes of the kernel stack space, and the average stack usage is less than 1,000 bytes [16]. This *reusability* of the kernel stack has resulted in good performance and high efficiency but also has contributed to unexpected leaks of stale values that are left on the kernel stack.

We use two real-world vulnerabilities to demonstrate a kernel information leak vulnerability through uses of uninitialized variables on the stack. Listing 1 shows an example of a kernel information leak caused by a use of uninitialized stack memory. In the `adjtimex` syscall, the `txc->tai` field is not initialized and is later used as an argument of `compat_put_timex`. Thus, the `compat_put_timex` function copies the `tai` field of the `txc` object to a local variable (`tx32` object), which is eventually copied to the user space and causes a kernel data leak. Listing 2 shows an another example of kernel information leak. Although all fields of the `map` object are initialized by the `rtnl_fill_link_ifmap` function,

```
1  /* file: kernel/time/time.c */
2  COMPAT_SYSCALL_DEFINE1(adjtimex, struct
       compat_timex __user *, utp)
3  {
4    struct timex txc; //stack object
5    int err, ret;
6
7    err = compat_get_timex(&txc, utp);
8    if (err)
9      return err;
10   ret = do_adjtimex(&txc);
11
12   //the above code does not write the 'tai' field
        of the 'txc' struct
13   err = compat_put_timex(utp, &txc);
14   ...
15 }
16
17 /* file: kernel/compat.c */
18 int compat_put_timex(struct compat_timex __user *
       utp, const struct timex *txc)
19 {
20   struct compat_timex tx32;
21   memset(&tx32, 0, sizeof(struct compat_timex))
22   tx32.modes = txc->modes;
23   ...
24   //copy the uninitialized data ('tai')
25   tx32.tai = txc->tai;
26
27   //kernel data leak to the user-space
28   if(copy_to_user(utp, &tx32, sizeof(struct
       compat_timex)))
29     return -EFAULT;
30   return 0;
31 }
```

Listing 1: A real-world vulnerability (CVE-2018-11508) in which an uninitialized field of the time struct in the stack caused the information leak.

```
1  /* file: net/core/rtnetlink.c */
2  static int rtnl_fill_link_ifmap(struct sk_buff *
       skb, struct net_device *dev)
3  {
4    //all fields in the map object are initialized
5    struct rtnl_link_ifmap map = {
6      .mem_start   = dev->mem_start,
7      .mem_end     = dev->mem_end,
8      .base_addr   = dev->base_addr,
9      .irq         = dev->irq,
10     .dma         = dev->dma,
11     .port        = dev->if_port,
12   };
13
14   //kernel data leak to the user-space
15   if(nla_put(skb, IFLA_MAP, sizeof(map), &map))
16     return -EMSGSIZE;
17   return 0;
18 }
```

Listing 2: A real-world vulnerability (CVE-2016-4486) which illustrates that padding bytes inserted by a compiler can bring the information leak.

the object still contains *uninitialized* 4 bytes padding generated by a compiler. Therefore, an unintended kernel data leak occurs when the stack object is copied to the user space by calling the nla_put function (on Line 14).

In both examples, the size of leaked data is 4 bytes, which is not enough to fully accommodate an 8-byte pointer value in 64-bit Linux. However, we will show in Section 5.4 that even a 4-byte leak is sufficient for leaking randomized kernel addresses.

2.2 Uninitialized Data in Linux Kernel Exploitation

As shown in Table 2, prior research work mostly focuses on controlled uses of uninitialized data in Linux kernels [11, 16, 25]. Our paper has a totally different goal: We focus on exploiting existing stack-based information leak vulnerabilities and converting them into high-impact vulnerabilities that leak sensitive data from the kernel. To the best of our knowledge, there is no prior research on the exploitation of stack-based information leak bugs in Linux kernel for leaking sensitive information.

2.3 Abusing Kernel Pointer Values

Bypassing KASLR. Commonly used in OS kernels, KASLR is a defense mechanism that randomizes the base address of the kernel (where the kernel code is loaded) at boot time. This technique was introduced to raise the bar of kernel memory corruption attacks (*e.g.*, buffer overflows and use-after-free attacks) and is one of the most effective defenses in modern OS kernels. Systems with KASLR enabled can successfully mitigate memory corruption attacks as long as the attacker cannot learn randomized kernel addresses through information disclosure or side channel leaks [13]. A kernel pointer leak will naturally lead to the bypass of KASLR, which we will detail next.

The Linux kernel on x86-64 architecture implements 6 bits of entropy for the kernel code. The address range of kernel text section is 1 GB (0xffffffff80000000 – 0xffffffffc0000000) and the base address of kernel text is aligned by 16 MB. Hence, there are 64 virtual memory addresses (1 GB ÷ 16 MB) where the kernel .text section can be loaded. Consequently, on a condition that we can leak a kernel pointer value pointing to a kernel function, we will be able to calculate the KASLR slide-byte by simply subtracting the 5th byte of the leaked pointer value from the 5th byte of kernel text section's start address. As an example, if the leaked kernel pointer value is 0xffffffffa9a72cc0, the KASLR slide-byte is 0xa9−0x80=0x29. Attackers can compute randomized addresses of all kernel functions by using the slide-byte.

Criteria	Our approach	Lu *et al.* [16]	Xu *et al.* [25]	Halvar Flake [11]
Types of unused memory targeted for generating exploits	Stack	Stack	Stack, Heap	Stack
Generating exploits for leaking sensitive data	✓	✗	✓	✗
Finding locations of uninitialized data	✓	✗	✗	✗
Reasoning about storing sensitive data at a given location	✓	✓	✗	✗

Table 2: Comparison of our proposed approach for uninitialized memory uses with the other approaches. Although there are several prior research works that focused on exploiting uninitialized data uses (such as uninitialized pointer dereferences), there has been no research effort on exploitations of stack-based information-leak bugs for leaking kernel pointer values.

Attacking the kernel stack. Another type of kernel pointer values that we attempt to leak are pointer values that point to the kernel stack. These kernel pointers can be used to identify where the kernel stack is. The location of the kernel stack must not be discovered by attackers because it is critical information that the attackers can use in their exploits to defeat KASLR and achieve arbitrary kernel code execution. For example, the kernel stack contains return addresses of kernel functions and values of the stack canary on which the entire stack overflow protection mechanism relies. Additionally, at the bottom of the kernel stack, the `thread_info` structure is stored (when `CONFIG_THREAD_INFO_IN_TASK` is disabled). This data structure includes architecture-specific thread-related information and a pointer to the `task_struct` that holds process-related information.

3 Attack Model

We assume that the attacker targets an x86-64 Linux system and tries to leak kernel pointer values that point to either kernel functions or the kernel stack. As previously discussed in Section 2.3, this step is very important for defeating modern kernel defenses, such as KASLR, before mounting future attacks.

To exploit information-leak bugs for leaking kernel pointer values, an in-depth analysis on the target kernel and the information-leaking bug is essential. Through this analysis, the attacker obtains critical information for exploiting the vulnerability, such as what types of kernel pointer values can be leaked, and where to place the kernel pointer values. We assume that the attacker has access to a local machine with the same Linux kernel and configuration as the target system, which the attacker can use to conduct the analysis and perform the attack before launching it on the target system. The attack should have full access to the local machine. We also assume that the attacker possesses the required exploit that triggers the information leak, which, at this moment, is likely to not leak any sensitive information on the kernel stack. With the analysis results, the attacker will generate exploits that can execute on the target system without the root privilege and reliably leak kernel pointer values.

4 Challenges in Exploitation

We demonstrated how kernel information leaks can occur via uninitialized stack uses with Listing 1 and Listing 2. However, simply triggering the vulnerabilities will most likely not copy any sensitive data from the kernel stack to the user space. Therefore, we must be able to manipulate data on the kernel stack and ensure kernel pointer values (or part of a kernel pointer value) are put in uninitialized variables on the stack. To this end, we must analyze each vulnerability and generate a proper exploit for it, which involves tackling the challenges that Lu, *et al.* previously discussed [16]. It is worth mentioning that our goal is different from theirs, and thus, we define a series of challenges that we must overcome to successfully leak kernel pointer values as follows.

C1: Computing the offset to uninitialized data from the kernel stack base.

The first challenge to leaking kernel pointer values is identifying the distance to an uninitialized memory cell from the base address of the kernel stack, which we term *leak offset*. Computing the leak offset allows us to find the exact location where kernel pointer values should be stored. We identify the leak offset through applying a new technique, called *footprinting*, on the kernel stack in Section 5.1.

C2: Storing kernel pointer values at a leak offset.

The next challenge is finding a way to place kernel pointer values at the specific leak offset. To achieve this goal, we propose two methods: (1) syscall enumeration with the help of the Linux Test Project (LTP) to find syscalls that can be used to store kernel pointer values at the leak offset (see Section 5.2), and (2) kernel stack spraying using the extended Berkeley Packet Filter (BPF) (see Section 5.3).

C3: Handling data leaks that are less than 8 bytes.

On a 64-bit Linux kernel, when a kernel data leak is larger than 8 bytes, we can obtain the value of the whole pointer. However, in many vulnerabilities, the size of memory leak is smaller than 8 bytes—we cannot obtain a complete pointer value. For handing such small leaks, we reason a possible range of the unleaked value through the guess and check method, by which we can identify the base address of the stack kernel. We discuss about the small leaks in Section 5.4.

Ideally, in addition to the above challenges, we should also prevent any future overwriting to kernel pointer values that we stored in the stack before the data is copied to the user space. This is to guarantee the successful exploitation of information-leak bugs. Unfortunately, there is no practical method to prevent such unexpected data overwriting without hijacking the control flow of the kernel on the target system. Thus, in this paper, we consider such cases where stored stack values are later overwritten before returning to user space to be *unexploitable*.

5 Exploiting Uninitialized Stack Variables

Our goal is to design a generic approach to exploit stack-based information-leak vulnerabilities for leaking kernel pointer values. In the rest of this section, we describe how we tackle the challenges that are represented in Section 4.

5.1 Computing the Leak Offset

We propose a novel technique, called byte-level stack footprinting, to identify the distance to an uninitialized memory address from the base address of the kernel stack. The mechanism is illustrated in Figure 1.

First, we write offset information to each byte of the stack from the base address by hooking a syscall. In 64-bit kernels, the kernel stack is 16-byte or 8-byte aligned at a new frame of a function starts and every pointer in the stack is stored at 8-byte aligned addresses. We store 1-byte offset information which starts from `0x0` to `0xff` in each byte for every 8 bytes. Therefore, even though with 1-byte information leak, we can identify exact offsets at which kernel pointer values should be stored to leak them. This mechanism allows us to footprint 2,024 bytes of the kernel stack. Even though we cannot footprint the entire kernel stack, 2,024 bytes are enough to deal with most syscalls (roughly 90% of syscalls only use less than 1,260 bytes of the stack).

Then we trigger an information-leak vulnerability. Because the offset information has been filled into the stack, we can directly check the offset. Lastly, we compute a leak offset by using the offset information from the kernel. For example, in Figure 1, the offset information copied from the kernel is `04`, and thus, we need to find kernel pointer values that can be stored at an offset ($Base - 24$).

5.2 Extensive Syscall Testing with the LTP

Once the leak offset has been identified, we need to find a syscall and its arguments that can be used to store a kernel pointer value at each leak offset. For fast and reliable testing, we leverage the Linux Test Project (LTP) which provides various tools and concrete test cases for syscalls [14]. We supplement three additional steps onto each syscall test case in LTP: (1) spraying the stack with a magic value; (2) finding kernel pointer values stored in the stack; and (3) recording context information.

Figure 2 shows how our syscall testing framework finds proper syscalls and arguments. Before executing a syscall, we fill the kernel stack with a magic value to detect data changes that are made by the execution of the syscall. Then we inspect the kernel stack from the base address to find kernel pointer values. To this end, we check every 8-byte from the stack base whether each 8-byte value is in the address range of the kernel stack or the kernel code region (the `.text` section). If we find any kernel pointer value that points to the kernel stack or kernel code, we record the name of the syscall with its arguments and pointer type. From the recorded information, we select proper context data (a syscall and arguments) that can store a kernel pointer value at a specific leak offset.

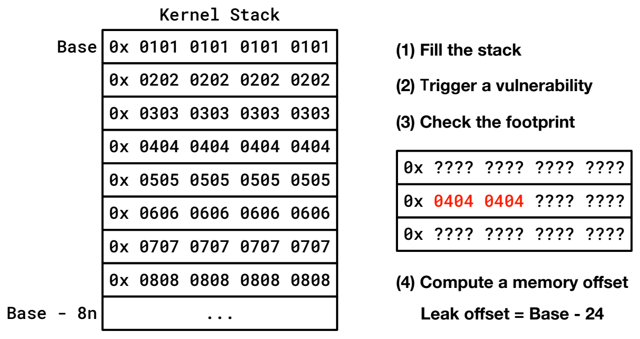

Figure 1: The footprinting mechanism of the kernel stack to compute a leak offset to an uninitialized memory area from the stack base. With this mechanism, we can footprint 2,024 bytes of the stack (about 90% of syscalls use less than only 1,260 bytes of the stack).

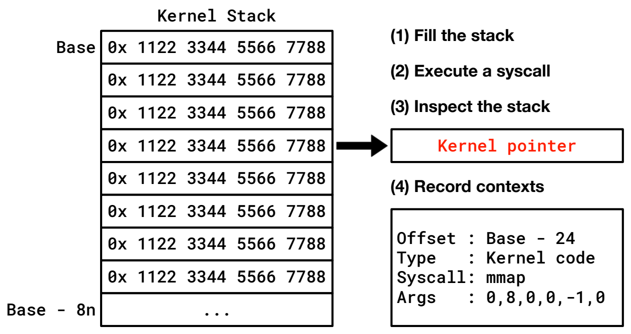

Figure 2: The routine of our syscall testing framework. We first fill the stack with a magic value. Then we execute a syscall and find kernel pointer values stored in the kernel stack. Lastly, we record the context information.

```c
static unsigned int __bpf_prog_run(void *ctx,
    const struct bpf_insn *insn)
{
  u64 stack[MAX_BPF_STACK / sizeof(u64)];
  // 512-byte stack for a BPF program

  u64 regs[MAX_BPF_REG], tmp;
  ...
```

Listing 3: The main function for executing a BPF program. It allocates the stack for BPF programs and execute them.

5.3 Stack Spraying via BPF

Designed to support filtering packets as requested by user-space applications, the extended Berkeley Packet Filter (BPF) is a virtual machine that resides inside the kernel [17]. The BPF virtual machine takes as input BPF programs (that use a special instruction set) when a user-space application attaches the BPF program onto any socket. Then, the BPF program executes when data passes through its attached socket and filters data as programmed.

BPF programs can use stack memory, which is allocated inside the kernel stack. Listing 3 shows the first part of the `bpf_prog_run` function, which shows that the stack of any BPF program is limited to 512 bytes. In the BPF virtual machine, there is a special register, `R10`, called the *frame pointer*. This register points to the top of the stack (the stack base) that a BPF program uses. Therefore, the frame pointer always points to a location on the kernel stack. We use this frame pointer and the location of the stack of a BPF program to spray the stack kernel.

With a carefully crafted BPF program, we can store the frame pointer value to the stack of a BPF program until the stack is full. In other words, we can store an address of the kernel stack up to 512 bytes inside the kernel stack. Additionally, the location of the `stack` local variable of the `bpf_prog_run` function changes depending on functions previously executed. Therefore, different execution paths *from* various syscalls transmitting data using a socket *to* the `bpf_prog_run` function can change the stack spraying range (discussed in Section 6.3 with a case study).

Listing 4 shows a part of a BPF program that sprays the kernel stack with the frame pointer. On Line 20, we copy the frame pointer (`R10`) to the `R3`. From Line 21, we spray the stack of a BPF program (kernel stack) with the frame pointer. It is worth noting that BPF virtual machine strictly restricts behaviors of a BPF program for preventing security issues by using the static verifier [7]. As examples of the restrictions for every BPF program, all memory access is bounded, there cannot be unreachable instructions, the frame pointer (`R10`) is a read-only register and so forth. However, the static verifier allows our BPF program to execute stack spraying. We manually inspected the verifier and could not find a rule for preventing spraying the frame pointer.

```c
1  #define BPF_MOV64_REG(DST, SRC)                 \
2  ((struct bpf_insn) {                            \
3    .code  = BPF_ALU64 | BPF_MOV | BPF_X,         \
4    .dst_reg = DST,                               \
5    .src_reg = SRC,                               \
6    .off   = 0,                                   \
7    .imm   = 0 })
8
9  #define BPF_STX_MEM(SIZE, DST, SRC, OFF)        \
10 ((struct bpf_insn) {                            \
11   .code = BPF_STX | BPF_SIZE(SIZE) | BPF_MEM,   \
12   .dst_reg = DST,                               \
13   .src_reg = SRC,                               \
14   .off   = OFF,                                 \
15   .imm   = 0 })
16
17 void stack_spraying_by_bpf()
18 {
19   struct bpf_insn stack_spraying_insns[] = {
20     BPF_MOV64_REG(BPF_REG_3, BPF_REG_10),
21     ...
22     BPF_STX_MEM(BPF_DW, BPF_REG_10, BPF_REG_3,
       -392),
23     BPF_STX_MEM(BPF_DW, BPF_REG_10, BPF_REG_3,
       -400),
24     BPF_STX_MEM(BPF_DW, BPF_REG_10, BPF_REG_3,
       -408),
25     ...
26   };
27   ...
```

Listing 4: A code snippet to perform kernel stack spraying using BPF. We can spray the frame pointer of a BPF program for 512 bytes on the kernel stack.

If we can store the frame pointer at a leak offset through the stack spraying, we will be able to figure out where the kernel stack is. Moreover, we can learn the memory layout of the kernel stack when a syscall executes based on the location of the kernel.

5.4 Handling Small Data Leaks

If an information-leak vulnerability leaks 8 bytes or more than 8 bytes of data, and we can store a kernel stack address at a leak offset through spraying the stack with a BPF program, it is possible to fully recover a kernel stack address. Unfortunately, the sizes of leaks of many stack-based information-leak vulnerabilities (roughly 60% of them) are smaller than 8 bytes [6]. Because the kernel stack is aligned by the size of a page (*e.g.*, 4KB by default), we need the most significant 52 bits of a kernel stack address (a 7-byte leak) to get the base of the kernel stack. Therefore, leaks that are smaller than 7 bytes cannot be directly used to reveal the kernel stack base.

To handle this problem, we investigated the static verifier of the BPF virtual machine to check if arithmetic operations on the frame pointer is possible. We found that some arithmetic operations (such as bitwise shift) are not possible, but `add` and `sub` can be used with arbitrary immediate values. If the BPF

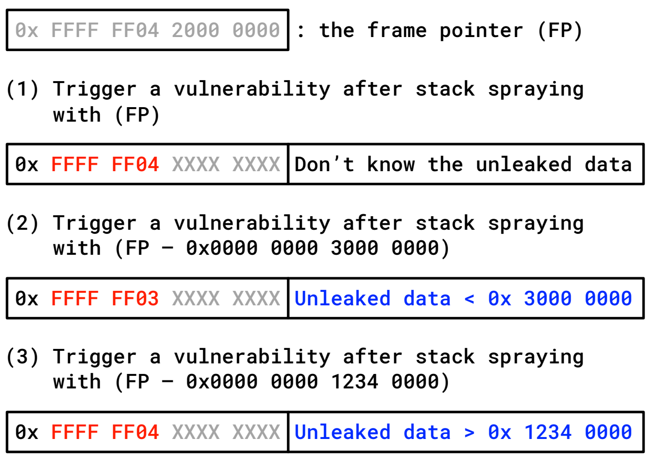
0x FFFF FF04 2000 0000
: the frame pointer (FP)
(1) Trigger a vulnerability after stack spraying with (FP)
0x FFFF FF04 XXXX XXXX
Don't know the unleaked data
(2) Trigger a vulnerability after stack spraying with (FP – 0x0000 0000 3000 0000)
0x FFFF FF03 XXXX XXXX
Unleaked data < 0x 3000 0000
(3) Trigger a vulnerability after stack spraying with (FP – 0x0000 0000 1234 0000)
0x FFFF FF04 XXXX XXXX
Unleaked data > 0x 1234 0000

Figure 3: The procedure for identifying a kernel stack address using the kernel stack spraying via a BPF program. By checking changes of the leaked data, we figure out possible ranges of the unleaked data until the stack base address is revealed.

allowed bit shifting operations on the frame pointer value, we would simply shift the frame pointer value so that unleaked data can be placed at leak offset. We can only execute `add` and `sub` operations on the frame pointer. However, these operations can be executed even if the result is beyond the range of the kernel stack. We also found that, after executing these arithmetic operations, the modified frame pointer value can be stored at the kernel stack.

By using this unrestricted behavior of a BPF program, we deal with such small leaks using the guess and check method to identify unleaked data of a kernel stack address, and, eventually, to reveal the layout of the kernel stack. This strategy requires manipulating the frame pointer value and check how known (leaked) data changes. Figure 3 illustrates how a 4-byte information leak vulnerability can be used to identify the base of the kernel stack by reasoning it. We first trigger a vulnerability after spraying the kernel stack with the frame pointer. Next, we execute an arithmetic operation (`add` or `sub`) on the frame pointer with an arbitrary immediate value. This modified frame pointer value is sprayed and we check the leaked data by triggering the vulnerability. As shown in Figure 3, when we sprayed (`FP−0x30000000`), the leaked data has changed from `0xffffff04` to `0xffffff03`, by which we can notice that the frame pointer value is smaller than `0xffffff0430000000`). We repeat this reasoning procedure until we can obtain the kernel stack base address: until the most important 52 bits of a kernel stack address is revealed.

We note that, a security patch was applied to the upstream Linux kernel at April 18th 2019 from the version 4.14.113[2] to restrict arithmetic operations on the frame pointer for unprivileged users so that the frame pointer value cannot go out of the stack region.

[2] https://lore.kernel.org/patchwork/patch/1063913/

6 Evaluation

We evaluate our proposed approach against real-world information leak vulnerabilities in Linux kernels that involve uses of uninitialized stack variables. In this section, we first present the implementation of our tool in Section 6.1, then present the evaluation results of our syscall-enumeration-based pointer finding approach (in Section 6.2), and finally present case studies of all five vulnerabilities that we evaluated against (in Section 6.3).

6.1 Implementation

We implemented an analysis tool, which consists of a shared library (KptrLib) and a loadable kernel module (KptrMod), to automatically find leak offsets (as described in Section 5.1). Then, we modified the Linux Test Project (LTP) to perform the three additional steps using KptrMod, as discussed in Section 5.2. We also implemented a tool for automatically spraying the kernel stack and handling small leaks with a BPF program (as described in Section 5.3 and Section 5.4).

Our tools can be easily used to analyze any given exploit that triggers a information-leak vulnerability to evaluate its exploitability regarding identifying the location of the kernel stack, leaking kernel pointers, and finally bypassing KASLR.

6.2 Finding pointers with the LTP framework

First, we evaluate the effectiveness of our syscall enumeration framework. To this end, we ran the modified LTP on Ubuntu 18.04 (with Linux kernel v4.15.0). For each kernel, we need to run the LTP framework once to record context information. Then, we can simply pick a context (a syscall and its argument) from the recorded data for storing a kernel pointer value at the identified leak offset.

Figure 4 illustrates that how many contexts (combinations of a syscall and its arguments) can store sensitive pointer values (pointing to the kernel code or stack) for each stack memory offset less than 2,298. The experimental results show that our modified LTP framework can find syscalls to store kernel pointer values at almost every stack offset when offsets are larger than `the stack base + 440`.

6.3 Case studies

To evaluate our approach, we select four CVEs and one fixed bug (which a CVE entry has not been assigned) in the Linux kernel and generate an exploit for each vulnerability according to our analysis results.

Why not analyze more CVEs? While we agree that analyzing more CVEs will help better demonstrate the generality and applicability of our proposed approach, during the evaluation of our approach, we realized that it is extremely

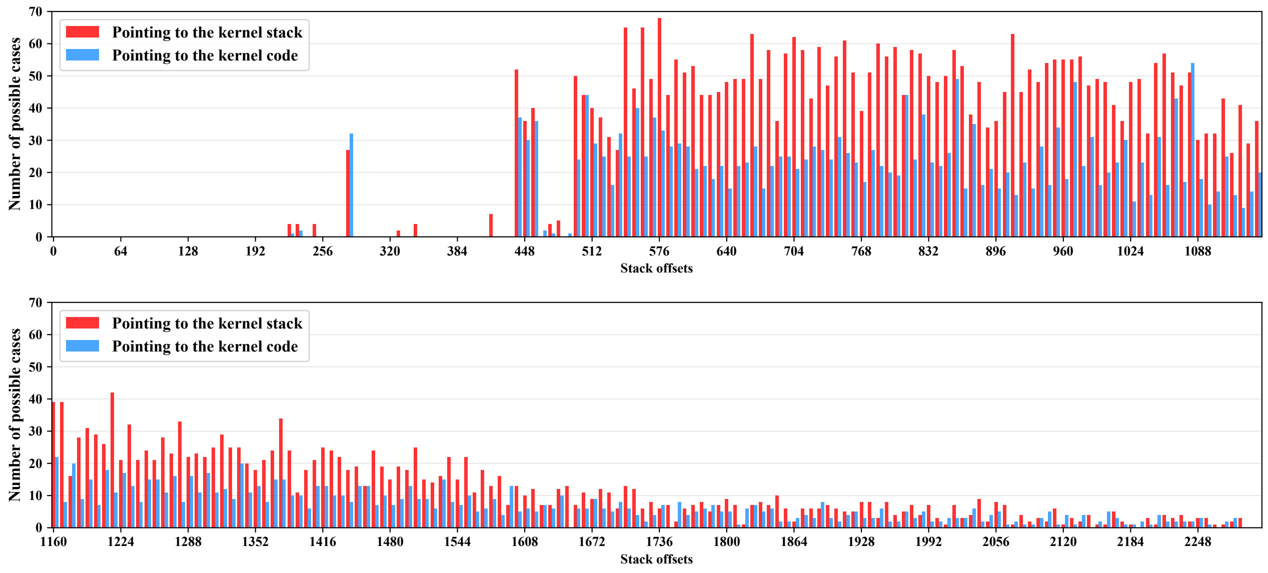

Figure 4: The experimental result of the modified LTP framework. We can find syscalls to store kernel pointer values at almost every stack offset, when offsets are larger than 440 bytes and smaller than 2,298 bytes, through the LTP framework.

time-consuming to develop exploits for these CVEs to reliably trigger the intended information-leaking bugs. Thus, we deem it infeasible to evaluate our approach on more CVEs for which no public exploits are available. We believe that these randomly selected CVEs are covering all scenarios that an attacker may face and are sufficient for demonstrating the generality of our proposed approach.

6.3.1 CVE-2018-11508

As we have shown in Section 2.1, this vulnerability is caused by an uninitialized stack variable (`tai` field of the `txc` struct). The CVSS score of this vulnerability is 2.1 [4]. We speculate that the impact of this vulnerability is deemed low because the size of information leak is only 4 bytes, which are not enough to host an entire pointer on 64-bit Linux systems.

After checking a leak offset through KptrLib and KptrMod, we found that this vulnerability leaks 5th byte to 8th byte of a pointer value. Also, we confirmed that a pointer value that points to the kernel text can be stored at the leak offset from the dataset recorded by the modified LTP in Section 6.2. Consequently, we can successfully get the KASLR slide from this vulnerability.

6.3.2 CVE-2016-4569 and fix 372f525

The CVSS score of CVE-2016-4569 is also 2.1. Our proposed approach successfully exploits this bug and identifies the KASLR slide.

The patch (fix 372f525) did not become an official CVE entry. In the commit message of the patch, a kernel developer mentioned that "*There should be no danger of breaking userspace as the stack leak guaranteed that previously meaningless random data was being returned.*[3]" Unfortunately, our proposed approach works against this bug and successfully identifies the KASLR slide. This demonstrates the necessity of our approach for proving the severity of information leak bugs in Linux kernels.

6.3.3 CVE-2016-4486

With this CVE (CVSS score 2.1), we show that how a 4-byte leak vulnerability can be exploited to identify the kernel stack base.

For spraying the kernel stack using the BPF program as in Listing 4, we first checked a leak offset of this vulnerability: the leak offset is *1,568*. We, then, executed the BPF program by calling the `sendmsg()` syscall. However, we could not clobber the offset because the BPF program sprays the kernel stack from offset 1,032 to 1,544 when we use the `sendmsg()` syscall. As we discussed in Section 5.3, there are various syscalls that can trigger the `bpf_prog_run()` function and each of them uses a different execution path to the BPF program runner—the stack spraying range is different based on the execution path. In this case, we found that we can clobber the leak offset through the `compat_sendmsg()` syscall by which we can spray the kernel stack from 1,064 to 1,576. Consequently, we could identify the kernel stack base with the guess and check method introduced in Section 5.4.

[3] https://github.com/torvalds/linux/commit/7c8a61d9ee

```
1  /*file: net/rds/recv.c */
2  void rds_inc_info_copy(struct rds_incoming *inc,
       struct rds_info_iterator *iter, __be32 saddr
       , __be32 daddr, int flip)
3  {
4    struct rds_info_message minfo;
5    minfo.seq = be64_to_cpu(inc->i_hdr.h_sequence);
6    minfo.len = be32_to_cpu(inc->i_hdr.h_len);
7    minfo.tos = inc->i_conn->c_tos;
8    if (flip) {
9      minfo.laddr = daddr;
10     minfo.faddr = saddr;
11     minfo.lport = inc->i_hdr.h_dport;
12     minfo.fport = inc->i_hdr.h_sport;
13   } else {
14     minfo.laddr = saddr;
15     minfo.faddr = daddr;
16     minfo.lport = inc->i_hdr.h_sport;
17     minfo.fport = inc->i_hdr.h_dport;
18   }
19   rds_info_copy(iter, *minfo, sizeof(minfo));
20   // The minfo struct is copied to the user-space
       with the uninitialized 'flag' field
```

Listing 5: The vulnerable function of CVE-2016-5244. The `rds_info_copy` function copies the `minfo` struct with `flag` field uninitialized.

6.3.4 CVE-2016-5244

The CVSS score of this vulnerability is 5.0, which is significantly higher than the other CVEs that we evaluate in this paper. Interestingly, we found that this one-byte leak vulnerability *cannot be exploited* through our analysis.

Listing 5 shows the vulnerable function where the `minfo` struct with the uninitialized `flag` field is copied to the user-space through the `rds_info_copy` function at Line 22. However, the leak offset of the uninitialized field (1 byte) always becomes `0` before the vulnerable function executes. Therefore, the vulnerability always leaks `0`, even though we can successfully store kernel pointer values at the leak offset.

6.3.5 Summary

In our evaluation, we analyzed four CVEs and one patch in the upstream Linux kernel as summarized in Table 3. We showed that our approach can effectively generate exploits. Additionally, the experimental results imply that our community is in need of a more accurate exploitability evaluation system for information leak bugs in Linux kernel so that security implications of bugs can be estimated more correctly.

7 Discussion

We discuss about limitations of this paper and possible mitigations against stack-based information-leak vulnerabilities.

Vulnerability	Leak Size	CVSS	Exploitation Result
CVE-2018-11580	4	2.1	Bypassed KASLR
CVE-2016-4569	4	2.1	Bypassed KASLR
CVE-2016-4486	4	2.1	Kernel stack base
Fixes: 372f525	4	N/A	Bypassed KASLR
CVE-2016-5244	1	5.0	Failed

Table 3: Summary of exploitation results of vulnerabilities. We analyzed 4 CVEs and 1 security patch which could not become a CVE entry.

7.1 Limitations

We showed that small leaks can be exploited to identify the KASLR slide (CVE-2018-11580) and the kernel stack base (CVE-2016-4486). Even though our approach to identifying the KASLR slide currently has no limitation in its usage, the BPF-based approach to reveal the stack base cannot be used in the Linux kernel from v.4.14.113 as in Section 5.4 (stack spraying is still possible). Therefore, we need a more general method to handle small leaks especially for revealing the stack base. To overcome this limitation, one possible strategy is to analyze the Linux kernel statically to find code gadgets which can modify the kernel stack with user-controlled data. We leave this limitation for future work.

Next, our approach analyzes information-leak vulnerabilities using programs that *can trigger* a vulnerability. Hence, we could not evaluate our approach in a large scale; Instead, we show the effectiveness of our approach against a limited number of vulnerabilities. This is mainly because generating such exploits manually is a time-consuming and complicated task. Even though we know which function has a vulnerability, we should find a proper context and create exploits to trigger it by manually analyzing the kernel source code. To enable large-scale experiments, our approach needs to be incorporated with emerging automatic exploit generation technologies such as FUSE [24].

7.2 Mitigating Uses of Uninitialized Memory

There are a couple of security features for uninitialized memory uses in the Linux kernel. STACKLEAK clears the kernel stack when syscalls return to the user-space, which was integrated into the Linux kernel upstream from v4.20 [20]. Recently, new configuration options, `CONFIG_INIT_ALL_MEMORY` and `CONFIG_INIT_ALL_STACK`, were introduced to force initialization of stack and heap variables [1]. In addition, many mitigation approaches have been proposed to prevent uninitialized memory uses. Peiró, *et al.* proposed a mechanism for detecting stack-based information-leak bugs of the Linux kernel through static data flow analysis [19]. Garmany, *et al.* have proposed another static data flow analysis framework that finds uninitialized stack memory uses after lifting binaries into an intermediate representation [12]. UniSan is a compiler-based approach to prevent information leaks caused by uninitialized read [15]. UniSan performs byte-level data flow analysis statically for OS kernels and instruments code to initialize data if it leaves kernel without initialization. The kernel memory sanitizer (KMSAN) is a tool to track uninitialized data to check whether the data leaves OS kernels or not, which can be utilized with fuzzers such as the syzkaller [5]. On the other hand, as a runtime defense system for OS kernels, kMVX has been proposed against information-leak vulnerabilities by leveraging the multi-variant execution [18]

8 Related work

Exploiting uninitialized memory uses. Albeit there have been research efforts on controlling uninitialized data to leverage it in other types of vulnerabilities, exploiting stack-based information-leak vulnerability to leak sensitive information such as pointer values pointing to the kernel stack or kernel code has not been explored yet [11, 16, 25].

Thomas Dullien (also known as Halvar Flake) proposed a search algorithm using call graphs for finding a function that can have a stack frame overlapping with the target memory address [11]. Lu, *et al.* proposed an automated method for writing arbitrary data to uninitialized stack variables through targeted stack spraying [16]. Xu, *et al.* showed common types of uninitialized uses and their potential threats by exploiting two uninitialized use vulnerabilities which can lead attackers to gain arbitrary kernel code executions in the macOS [25].

Automating kernel exploitation. Automated kernel exploit generation is a demanding task. In addition, even determining the exploitability of bugs requires significant manual efforts. Security researchers have been attempting to address these problems. FUZE [24] proposed to identify useful system calls for kernel use-after-free exploitations by leveraging fuzzing and symbolic execution techniques. KEPLER [23] showed a code-reuse exploit approach that converts a user-provided control-flow hijacking primitives into arbitrary stack overflows, and thus, it bootstraps return-oriented programming (ROP) payload. Chen *et al.* [9] proposed static and dynamic analysis methods to find useful data structures for use-after-free exploitations in the Linux kernel.

9 Conclusion

In this paper, we proposed a generic approach to exploit uses of uninitialized stack data in Linux kernels to leak pointer values that are pointing to either kernel functions or to the kernel stack. These leaked pointer values can then be used to defeat KASLR and mount future attacks against Linux kernels. Our evaluation results show that we can effectively analyze and exploit stack-based information-leak vulnerabilities through the proposed approach. Our proposed approach exposes the actual exploitability and severity of information disclosure bugs in Linux kernels and will raise awareness of the community on the security impact of these bugs. We expect our findings will help adjust CVSS scoring for information leak bugs inside Linux kernels.

Acknowledgment

We would like to express our gratitude to the anonymous reviewers for their thoughtful reviews.

This material is based upon work supported in part by Samsung Research, Samsung Electronics, the National Science Foundation (NSF) under Grant Number 1703644, the Defense Advanced Research Projects Agency (DARPA) under agreement number HR001118C0060 and FA875019C0003, the Office of Naval Research (ONR) KK1847, the Global Research Laboratory Program through the National Research Foundation of Korea funded by the Ministry of Science and ICT under Grant NRF-2014K1A1A2043029, and a grant from the Center for Cybersecurity and Digital Forensics (CDF) at Arizona State University.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

References

[1] *Automatic variable initialization*. https://reviews.llvm.org/D54604.

[2] *CVE (Vulnerability) Details: CVE-2016-4486*. https://www.cvedetails.com/cve/CVE-2016-4486.

[3] *CVE (Vulnerability) Details: CVE-2016-4569*. https://www.cvedetails.com/cve/CVE-2016-4569.

[4] *CVE (Vulnerability) Details: CVE-2018-11508*. https://www.cvedetails.com/cve/CVE-2018-11508.

[5] *KernelMemorySanitizer, a detector of uses of uninitialized memory in the Linux kernel*, (accessed Feb 2nd, 2020). https://github.com/google/kmsan.

[6] *Linux Kernel: Vulnerability Statistics*, (accessed Feb 2nd, 2020). https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[7] *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*, (accessed Feb 2nd, 2020). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[8] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, Shanghai, China, July 2011.

[9] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 1707–1722, London, UK, October 2019.

[10] Kees Cook. *Kernel exploitation via uninitialized stack*, 2011. https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf.

[11] Halvar Flake. *Attacks on uninitialized local variables*, 2006. https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf.

[12] Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz. Static Detection of Uninitialized Stack Variables in Binary Code. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, pages 68–87, Luxembourg, September 2019.

[13] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 380–392, Vienna, Austria, October 2016.

[14] Paul Larson. Testing linux with the linux test project. In *Proceedings of the Linux Symposium (OLS) 2002*, Ottawa, Canada, June 2002.

[15] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 920–932, Vienna, Austria, October 2016.

[16] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[17] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993*, pages 259–270, San Diego, CA, January 1993.

[18] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 559–572, Providence, RI, April 2019.

[19] Salva Peiró, M Muñoz, Miguel Masmano, and Alfons Crespo. Detecting stack based kernel information leaks. In *Proceedings of the International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*, pages 321–331, Bilbao, Spain, June 2014.

[20] Alexander Popov. STACKLEAK: A Long Way to the Linux Kernel Mainline. In *Linux Security Summit 2018*, Vancouber, Canada, August 2018.

[21] Alexander Potapenko. Dealing with Uninitialized Memory in the Kernel. In *Linux Security Summit Europe 2019*, Lyon, France, October–November 2019.

[22] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, Seoul, South Korea, July 2012.

[23] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 1187–1204, Santa Clara, CA, August 2019.

[24] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, pages 781–797, Baltimore, MD, August 2018.

[25] Zhenquan Xu, Gongshen Liu, Tielei Wang, and Hao Xu. Exploitations of uninitialized uses on macos sierra. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, Canada, August 2017.