CS202 eBPF Security Project Report

Group 1

Xiaopei Wang xwang605
Milan Maragiri mmara031

1 Introduction

In this project, we explored whether SafeBPF could prevent CVE-2017-16995, a known vulnerability in the Linux eBPF subsystem. Since SafeBPF could not be patched onto Linux 4.11.110 (one of the versions affected by this CVE), we attempted to recreate the vulnerability in Linux 6.3.8 by exploring different methods to bypass or skip the verifier.

Our results indicate that while skipping the verifier may also bypass some critical functionality—leading to kernel crashes or preventing map writes—SafeBPF successfully intercepted an invalid page fault. This suggests that SafeBPF has the potential to mitigate the CVE if properly deployed.

1.1 eBPF Framework and Sockets

The extended Berkeley Packet Filter (eBPF) is a powerful in-kernel virtual machine that enables the execution of sandboxed programs within the Linux kernel. eBPF is widely used for networking, tracing, and security applications. One of its key features is the ability to attach programs to sockets, enabling efficient packet filtering and manipulation without requiring kernel modifications.

eBPF programs are loaded and verified by the kernel, ensuring they adhere to security constraints before execution. The verifier enforces restrictions to prevent unsafe memory access and control flow manipulation, limiting the risk of kernel exploits.

1.1.1 Testing a Simple eBPF Program

To validate our test environment, we implemented and executed a basic eBPF program that reads and writes data. The test confirmed that eBPF functions correctly on Linux 6.3.8, providing a baseline for further experimentation.

1.2 CVE-2017-16995 Overview

We first introduce how this CVE works. As we know, the verifier attempts to simulate the execution of a BPF program to determine whether it is malicious. This CVE exploits a bug in the verifier that allows premature termination of the verification process.

```
register uint64_t r9 = (uint32_t)0xFFFFFFFF;
if (r9 == -1) {
    exit(); // Valid branch
} else {
    // Malicious branch
    // Execute malicious code
}
```

Because of a type-conversion bug in the verifier, it mistakenly assumes that the exiting valid branch must be taken. As a result, it prematurely terminates the verification process when it believes the program will exit. However, in actual execution, the condition `r9 == -1` will not be satisfied since `0x00000000FFFFFFFF (r9)` $\neq$ `0xFFFFFFFFFFFFFFFF (-1)`. Consequently, the verifier is bypassed, leading to an unsafe kernel state.

Then we introducing the exploit code used in our experiment, we borrowed the implementation from `https://github.com/bsauce/kernel-exploit-factory/blob/main/CVE-2017-16995/exp.c`. The verifier prematurely terminates execution on the valid branch, making it possible to execute malicious code. Below is a brief explanation of how the malicious branch achieves privilege escalation.

This exploit BPF communicates with user space through a map containing three entries. The malicious code first reads values from these three map entries and stores them in three registers. Based on the values of these registers, the BPF program can perform different actions as specified by the attacker. It provides three options: leaking `r10`, arbitrary read, and arbitrary write.

The key aspect is the "leak `r10`" option, which writes the value of `r10` (the frame pointer, corresponding to `rbp`) to the map, making it accessible to the user. Ultimately, the attacker can obtain the `task_struct` address and instruct the BPF program to overwrite the credential structure by writing `0` to the credential field within `task_struct`, thereby achieving privilege escalation.

In conclusion, the exploit is effective because the eBPF program is designed to read and write directly to kernel address space and also bypass the verifier. It communicates with a user-space program via an eBPF map, allowing the user to control the eBPF program and manipulate kernel memory, potentially leaking or modifying sensitive information.

1.3 SafeBPF and Its Defense Mechanism

SafeBPF addresses security risks in eBPF by implementing dynamic sandboxing, which isolates eBPF programs from the kernel to prevent unauthorized memory access. It employs two key techniques:

1. Software-based Fault Isolation (SFI): Restricts memory access within predefined sandbox boundaries using address masking.
2. Hardware-assisted Memory Tagging (MTE): Uses ARM's Memory Tagging Extension to dynamically enforce spatial memory safety.

By confining all eBPF memory operations within a controlled environment, SafeBPF prevents exploits that bypass static verification checks, such as those used in CVE-2017-16995, which allowed attackers to escalate privileges. In our project, we used Software-based Fault Isolation (SFI) to evaluate whether SafeBPF could prevent CVE-2017-16995.

2 Environment setup

To conduct our tests, we set up a QEMU-based virtualized Linux environment for x86 architecture that allowed us to easily switch between kernel versions while maintaining a consistent filesystem.

1. Linux Kernel 6.3.8 was downloaded from the Linux Kernel Archives for the x86 architecture. Source: Linux Kernel Archives
2. We used an existing CVE-2017-16995 exploit from GitHub: Kernel Exploit Factory
3. Linux Kernel 4.4.110 was downloaded to test the functionality of the exploit code. Source: Linux Kernel Archives

4. eBPF subsystem is enabled and the kernel is compiled using the following commands

```
sed -i 's/# CONFIG_BPF_SYSCALL is not set/CONFIG_BPF_SYSCALL=y/' .
    config
sed -i 's/# CONFIG_BPF_JIT is not set/CONFIG_BPF_JIT=y/' .config

make oldconfig
make -j$(nproc)
```

5. We also disabled certain memory safety features in the linux kernel such as KASLR (Kernel Address Space Layout Randomization) and KPTI (Kernel Page Table Isolation) that were introduced in the newer version of Linux by using the following commands

```
sed -i 's/CONFIG_PAGE_TABLE_ISOLATION/CONFIG_PAGE_TABLE_ISOLATION=n/'
    .config
sed -i 's/CONFIG_RANDOMIZE_BASE/CONFIG_RANDOMIZE_BASE=n/' .config
```

6. Qemu Setup

   - We used the Debian 12 (Bookworm) cloud image as our root file system
   - Source: Debian Cloud Image
   - The kernel was booted dynamically using QEMU's -kernel option

```
qemu-system-x86_64 -enable-kvm \
    -m 8G  \
    -smp 2  \
    -kernel ./bzImage    \
    -append "console=ttyS0 root=/dev/sda1 rw earlyprintk=serial net
       .ifnames=0 kaslr no_hash_pointers idle=halt" \
    -drive file=./debian-12-nocloud-amd64.qcow2,format=qcow2 \
    -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 \
    -net nic,model=e1000 \
    -cpu host \
    -nographic  \
    -pidfile vm.pid
```

   - -m 8G → Allocates 8GB RAM.
   - -smp 2 → Uses 2 CPU cores.
   - -kernel bzImage → Boots with our custom Linux 6.3.8 kernel.
   - -drive file=debian-12-nocloud-amd64.qcow2 → Loads Debian 12 as the root filesystem.
   - -net=allows internet access to the linux kernel

7. Once the system was set up, we also compiled a Linux 6.3.8 kernel with SafeBPF patch. SafeBPF patch source: safeBPF GitHub

8. The summary is shown in Table 1

Component	Details
Kernel	Linux 6.3.8 and Linux 4.4.110
Exploit	CVE-2017-16995 exploit from GitHub
Virtualization	QEMU (for switching kernels easily)
Filesystem	Debian 12 (Bookworm) cloud image
Test Patching	Applied safeBPF patch to test behaviour

Table 1: Key components used in our experiments

3 Experiments

3.1 Preparations

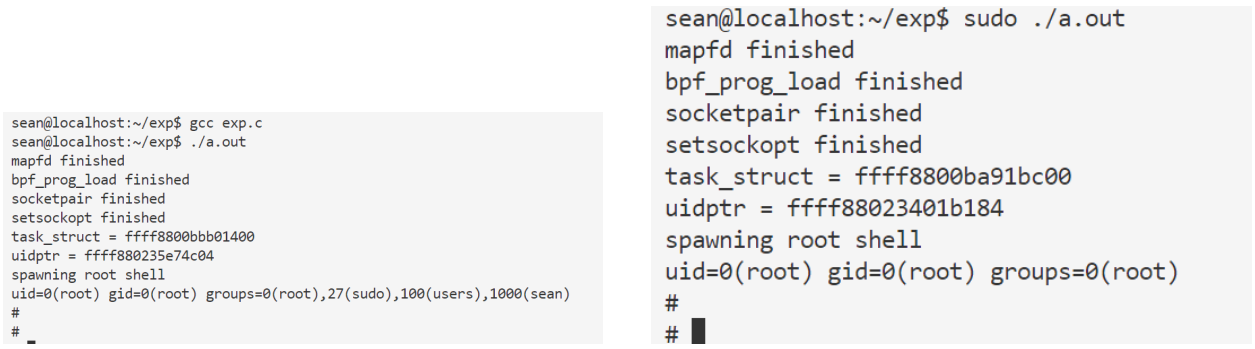
3.1.1 Exploit on Linux version 4.4.110

To execute the exploit code, we first needed to determine the `CRED_OFFSET` for the specific kernel version. We compiled the source code with full debugging information and then used `pahole` to extract the corresponding credential offset. As shown in Listing 1, the `cred` offset for Linux version 4.4.110 is `1456`.

Since the credential offset varies across different kernel versions, we always update the offset value in the exploit code accordingly when conducting experiments on different Linux kernel versions.

```
[xwang605@xe-07 ~]$ pahole -C task_struct /home/csgrads/xwang605/linux
   -4.4.110/linux-4.4.110/vmlinux | grep cred
        const struct cred   *       ptracer_cred;          /*   1440      8 */
        const struct cred   *       real_cred;             /*   1448      8 */
        const struct cred   *       cred;                  /*   1456      8 */
```

Listing 1: Extracting `cred` Offset Using `pahole`



(a) Root access gained as a normal user.

(b) Root access gained under `sudo` privileges.

Figure 1: Exploit success in escalating privileges with and without `sudo` on Linux 4.4.110.

As shown in Figure 1, a normal user has successfully escalated privileges to root. This indicates that the exploit can work if the BPF program can bypass the verifier successfully.

3.1.2 Patching SafeBPF on Linux version 6.3.8

Since the source code of the SafeBPF patch is only available for Linux version 6.3.8, we use this version for our subsequent experiments. We conduct our tests on two kernel versions: the original Linux 6.3.8 and the SafeBPF-patched Linux 6.3.8.

3.2 Bypassing the verifier by reverting the CVE fix

To test whether we could bypass the verifier by simply reintroducing the CVE, we reverted the fix in commit 95a762e2c8c9, making the vulnerability reappear in the fix-reverted kernel.

After running the exploit code on it, we observed that we still could not pass the verifier when attempting to load the BPF program as a normal user without `sudo` privileges. The verifier reported an error regarding `R7` as an "invalid memory access: scalar." However, when loading the program with `sudo`, we observed different behavior: if the fix was reverted, the verifier could be bypassed, as no verifier logs or "Permission denied" errors appeared. In contrast, a normal Linux kernel with the fix would still reject the program, even when loaded with `sudo` privileges. Same situation happens when SafeBPF patch applied. The result with different combinations are shown in Table 2

Patch Configuration	**Without** `sudo`	**With** `sudo`
Reverted Fix	Verifier error: permission denied	Verifier bypassed
With Fix	Verifier error: permission denied	Verifier error: permission denied
SafeBPF Patch + Reverted Fix	Verifier error: permission denied	Verifier bypassed
SafeBPF Patch + Fix	Verifier error: permission denied	Verifier error: permission denied

Table 2: Effect of different patch configurations on malicious BPF program verification

To determine why the exploit still could not pass the verifier for normal users even after reverting the fix, we analyzed the verifier's log. We discovered that the verifier rejected the program because the malicious branch was also verified, resulting in an error related to an arbitrary read instruction.

However, when executing the exploit with `sudo` on a kernel with the reverted fix, we were able to bypass the verifier, possibly due to premature termination. This suggests that loading the BPF program with `sudo` somehow alters the verifier's behavior in a way that is not yet fully understood.

Upon further examination of the verifier source code, we found that the verifier does indeed behave differently depending on whether the BPF program is loaded with `sudo` privileges. There are two key variables: `env->bpf_capable` and `is_priv`. These variables are assigned values by the function `bpf_capable()`, which returns `1` under `sudo` privileges and `0` under normal user conditions. These two variables influence certain branch conditions in the verifier, meaning that its behavior is privilege-dependent. We can reasonably assume that the verifier is less strict under `sudo` privileges, which aligns with the scenario observed in Table 2, where running the exploit as a `sudo` user resulted in bypassing the verifier.

Worth mentioning that for Table 2, the verifier error log remains the same across different configurations. The detailed verifier log can be found in Appendix A.

To run the exploit program as a normal user, bypass the verifier, and evaluate whether SafeBPF can effectively prevent the CVE, we conducted a series of experiments. In these tests, we modified the verifier code to manually skip certain checks, simulating a scenario where our malicious eBPF program successfully bypasses the verifier.

Another important observation is that, although we were able to bypass the verifier with sudo, we could not achieve privilege escalation. The exploit code failed at the first step: attempting to fetch the value in `r10` (which corresponds to `rbp`, the fp: frame pointer) and store it in the user-accessible map. The actual retrieved value was `0`, meaning that the subsequent steps: modifying credentials and obtaining root access could not be executed.

3.3 Manually skipping the verifier by returning on the valid branch.

As we know, the key mechanism of this CVE is the premature termination of the verifier. The previous experiment suggests that there may be additional detection mechanisms in the verifier that can still run the verification on the malicious branch. To better simulate the exact behavior of this CVE, we introduced an early return when encountering the exit instruction on the valid branch of the BPF program, replicating the conditions under which this vulnerability is triggered (early termination of verifier).

In the function `do_check()`, we added an unconditional `break` statement under the `process_bpf_exit` flag. Previously, when a BPF branch currently being parsing by the verifier encountered an exit instruction, execution would jump to this flag. Under this flag, the loop would terminate only if there were no other branches left to parse. By introducing this unconditional `break`, we were able to simulate the early termination of the verifier.

Then, we tested our exploit code on both the SafeBPF-patched and non-patched kernels.

Patch Configuration	**Without** `sudo`	**With** `sudo`
Non-Patched Kernel	Kernel crash	Verifier bypassed, but frame pointer = 0
SafeBPF-Patched Kernel	Kernel crash	Verifier bypassed, but frame pointer = 0

Table 3: Result of running exploit code on kernels of verifier that was modified to return on the valid branch

As shown in Table 3, when running the exploit without `sudo`, the kernel crashed by having a CPU stall. Conversely, when executing the exploit with `sudo`, the kernel did not crash and the verifier was bypassed, showing the same result on Table 2. This behavior was observed consistently across both patch configurations. The full kernel log of the stall can be found in Appendix B.

At this point, we were still unable to draw a definitive conclusion regarding why the kernel crashes without `sudo` privileges. We also suspected that our modification to the verifier—specifically, the unconditional `break` statement might have introduced a bug in the kernel, causing the crash.

In our previous experiment, we observed that the verifier behaves differently under `sudo` privileges. Therefore, it is possible that `sudo` bypasses certain checks in the verifier, which prevents the bug introduced by our `break` statement from triggering a crash.

Thus, the experiments in the following subsections were conducted by attempting to manipulate different methods of bypassing the verifier to further narrow down the cause of the kernel crash and attempt to achieve privilege escalation by further modifying the verifier.

3.4 Manually skipping the verifier by returning at the beginning

The assumption was that the previous approach had partially executed code, leading to an incomplete state that may have caused the kernel crash. To address this, we modified the verifier to return immediately upon entering the function, instead of exiting at the valid branch. This effectively bypassed all verifier logic and forced an immediate success response.

The changes made to the verifier code in the function `bpf_check()` are shown in Listing2.

```
    is_priv = bpf_capable(); // original verifier code
    printk("SKIP␣bpf␣verifier:␣is_priv:␣␣%d\n",is_priv);
    return 0; // force early return
```

Listing 2: Changes made to retun at the biginning

After making changes to the verifier, we run the exploit code and observed the following results in Table 4. The kernel logs of sudo and non sudo remains the same for same patch configuration, corresponding kernel

Patch Configuration	**Without** `sudo`	**With** `sudo`
Non-Patched Kernel	Page Fault	Page Fault
SafeBPF-Patched Kernel	NULL pointer de-reference	NULL pointer de-reference

Table 4: Result of running exploit code on kernels of verifier that was modified to return at the beginning

logs can be found in Appendix C.

Running the exploit without SafeBPF results in a page fault being recorded in the kernel logs. However, when executing the exploit with SafeBPF enabled, a different log message appears. Based on the sandbox mechanism of SafeBPF, the invalid memory access may be masked as a NULL pointer.
In all cases, these errors do not crash the kernel and are recoverable, either the kernel terminates the process automatically or we can manually stop it using `Ctrl + C`.

By further analyzing the kernel logs of page faults (shown in Appendix C), we concluded that the page fault occurs during the execution of the malicious BPF program. This conclusion is supported by our observation of the kernel log entry:

```
RIP: 0010: sk_filter_trim_cap +0xe0 /0x230
```

which indicates that the crash occurs within the socket-related function. Since our BPF program is bound to a socket, it gets executed every time data is written to that socket. The kernel consistently reports a page fault whenever we write to the socket, confirming that the fault occurs precisely when the BPF program is triggered.

However, crashing the kernel was not the intended behavior of the exploit code, and we were unable to achieve privilege escalation since the kernel would terminate the process upon detecting the fault.

By further examining the verifier code, we found that returning at the beginning of the function was not the correct approach. In Linux version 4.4.110, we identified a function in the verifier named `replace_map_fd_with_map_ptr`, which, in Linux version 6.3.8, is referred to as `resolve_pseudo_ldimm64` and performs a similar function. If a BPF program uses a map, it does not initially know the actual map pointer at creation time. Instead, the BPF program inserts a placeholder at the corresponding position in the bytecode, and the verifier dynamically replaces this placeholder with the actual map pointer.

If we return at the very beginning of the verification process, we inadvertently skip this critical initialization step, causing the map access to be invalid since it was never replaced by a real map pointer. This results in a page fault or a NULL pointer dereference.

We confirmed that the fault was due to skipping the initialization process at the beginning because we were able to reproduce the same page fault by introducing an early return in the verifier in Linux version 4.4.110.

Thus, the following experiment was conducted in an attempt to find the correct way to bypass the verifier while still preserving its functionality.

3.5 Manually skipping the verifier by hard-coding all return paths to success

In this experiment, we modified `verifier.c` by changing all error return values in `bpf_check()` to a success value of 0. This approach ensures that all required functionalities of the verifier are executed while still forcing it to approve every eBPF program, regardless of its actual safety. However, this modification resulted in a kernel crash, exhibiting the same behavior observed in Section 3.3.

To investigate the cause of the crash, we ran a simple, non-malicious eBPF program that only performs legitimate reads from an eBPF map. With the modified verifier in place, we observed that the kernel crashed in non-`sudo` mode, whereas it remained stable when executed with `sudo` privileges. We observed the same behavior as in Table 3. Results are summarized in Table 5.

The results holds for both the SafeBPF-patched and the original Linux kernels. And the crashing kernel log is the same as in Section 3.3 shown in Appendix B.

Verifier Configuration	**Without `sudo`**	**With `sudo`**
Modified Verifier (Skipping Checks)	Kernel crash	No crash
Original Verifier	No crash	No crash

Table 5: Results of executing a normal BPF program on kernels where the verifier was modified to skip verification by returning in a valid branch or always returning success.

At this point, we can confirm that neither mechanism—skipping the verifier by returning in a valid branch or always returning success—preserves its full functionality, as the normal kernel with the original verifier does not crash under any circumstances. Additionally, we confirm that the kernel crash occurs due to memory access to a map when the verifier is skipped, which may result in the map not being set up correctly.

Further investigation is required to determine the underlying reason for this discrepancy, particularly why the kernel behaves differently between privileged and unprivileged execution with the modified verifier. At this stage, we can only hypothesize that, although map access with the modified verifier is problematic, having `sudo` privileges might prevent the CPU from stalling under certain conditions.

We can reasonably eliminate the possibility that the modified verifier correctly sets up the map under `sudo`, as this would have allowed us to achieve root access by interacting with the BPF program through map accesses. However, in reality, the user space always reads `0` from the map, indicating that the map was not properly initialized. This observation suggests that a more plausible explanation is that "accessing invalid memory as `sudo` may not cause a CPU stall, whereas doing so as an unprivileged user would."

4 Conclusion

It may not be possible to both skip the verifier and allow it to perform its intended function. As concluded from Section 3.5, certain functionalities—such as replacing the placeholder map file descriptor in the bytecode with the actual map descriptor, which we missed in Section 3.4—may be essential for proper operation. The verifier is designed not only to validate program correctness but also to perform critical transformations necessary for program execution. By manually skipping the verifier, these transformations may also get bypassed, which can lead to unintended consequences, such as being unable to write to the map or even triggering a kernel crash. Moreover, there may be additional verifier-assisted functionalities that remain unidentified, further complicating any attempt to skip the verifier while maintaining kernel stability.

A comprehensive understanding of the SafeBPF code would be necessary to patch it on an older kernel where the exploit is still present. This would allow us to test this and other CVEs to assess SafeBPF's strengths

and weaknesses. Alternatively, a thorough understanding of the verifier would be required to successfully bypass it with malicious code, simulating the exploit on a newer kernel and analyzing SafeBPF's effectiveness.

Our experiment in Section 3.4 demonstrates that, instead of the kernel encountering a page fault, SafeBPF successfully detects the invalid access and logs it as a null pointer dereference. This indicates that SafeBPF has the potential to mitigate the CVE if properly deployed on newer kernels.

Although we were ultimately unable to achieve privilege escalation, we gained valuable insights throughout the experimentation process.

A Verifier Log for Table 2

The following is the verifier log observed during our experiment on Table 2:

```
Verifier log:
0: R1=ctx(off=0,imm=0) R10=fp0
0: (b4) w9 = -1                       ; R9_w=P-1
1: (55) if r9 != 0xffffffff goto pc+2 4: R1=ctx(off=0,imm=0) R9_w=Pscalar
   () R100
4: (18) r9 = 0x0                      ; R9_w=map_ptr(off=0,ks=4,vs=8,imm
   =0)
6: (bf) r1 = r9                       ; R1_w=map_ptr(off=0,ks=4,vs=8,imm
   =0) R9_)
7: (bf) r2 = r10                      ; R2_w=fp0 R10=fp0
8: (07) r2 += -4                      ; R2_w=fp-4
9: (62) *(u32 *)(r10 -4) = 0          ; R10=fp0 fp-8=0000????
10: (85) call bpf_map_lookup_elem#1   ; R0=map_value_or_null(id=1,off=0,ks
   =4,vs)
11: (55) if r0 != 0x0 goto pc+1 13: R0=map_value(off=0,ks=4,vs=8,imm=0) R9
   =map_?
13: (79) r6 = *(u64 *)(r0 +0)         ; R0=map_value(off=0,ks=4,vs=8,imm
   =0) R6_)
14: (bf) r1 = r9                      ; R1_w=map_ptr(off=0,ks=4,vs=8,imm
   =0) R9=)
15: (bf) r2 = r10                     ; R2_w=fp0 R10=fp0
16: (07) r2 += -4                     ; R2_w=fp-4
17: (62) *(u32 *)(r10 -4) = 1         ; R10=fp0 fp-8=mmmm????
18: (85) call bpf_map_lookup_elem#1   ; R0=map_value_or_null(id=2,off=0,ks
   =4,vs)
19: (55) if r0 != 0x0 goto pc+1 21: R0=map_value(off=0,ks=4,vs=8,imm=0) R6
   =Psca?
21: (79) r7 = *(u64 *)(r0 +0)         ; R0=map_value(off=0,ks=4,vs=8,imm
   =0) R7_)
22: (bf) r1 = r9                      ; R1_w=map_ptr(off=0,ks=4,vs=8,imm
   =0) R9=)
23: (bf) r2 = r10                     ; R2_w=fp0 R10=fp0
24: (07) r2 += -4                     ; R2_w=fp-4
25: (62) *(u32 *)(r10 -4) = 2         ; R10=fp0 fp-8=mmmm????
26: (85) call bpf_map_lookup_elem#1   ; R0=map_value_or_null(id=3,off=0,ks
   =4,vs)
27: (55) if r0 != 0x0 goto pc+1 29: R0=map_value(off=0,ks=4,vs=8,imm=0) R6
   =Psca?
29: (79) r8 = *(u64 *)(r0 +0)         ; R0=map_value(off=0,ks=4,vs=8,imm
   =0) R8_)
30: (bf) r2 = r0                      ; R0=map_value(off=0,ks=4,vs=8,imm
   =0) R2_)
31: (b7) r0 = 0                       ; R0_w=P0
32: (55) if r6 != 0x0 goto pc+3       ; R6=P0
33: (79) r3 = *(u64 *)(r7 +0)
R7 invalid mem access 'scalar'
processed 33 insns (limit 1000000) max_states_per_insn 0 total_states 3
   peak_st2
```

```
bpf_prog_load failed: Permission denied (errno: 13)
error: Permission denied
```

The exploit code we used places a user-controlled memory address into register `R7`:

```
r7 = *(u64 *)(r0 + 0) ; R0 = map_value(off=0,ks=4,vs=8,imm=0) R7_w =
    scalar()
```

Then, it attempts to access the value stored at address `R7`:

```
r3 = *(u64 *)(r7 + 0)
```

The verifier detects this malicious behavior (arbitrary read) and rejects the BPF program accordingly.

B Kernel Log for CPU stall

```
[   40.302149] rcu: INFO: rcu_preempt self-detected stall on CPU
[   40.302620] rcu:     0-....: (20995 ticks this GP) idle=dbac/1/0
   x4000000000000000 softirq=4046/4046 fqs=5249
[   40.303297] rcu:     (t=21002 jiffies g=1657 q=1445 ncpus=2)
[   40.303682] CPU: 0 PID: 187 Comm: a.out Not tainted 6.3.8-g7a95096a8a87
   -dirty #16
[   40.304216] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS
    rel-1.16.2-0-gea1b7a073390-prebuilt.qemu.org 04/01/2014
[   40.305022] RIP: 0010:0xffffffffa000949f
[   40.305312] Code: cc cc cc f3 0f 1e fa 0f 1f 44 00 00 66 90 55 48 89 e5
    f3 0f 1e fa 41 57 41 bf ff ff ff ff 49 83 ff ff 75 06 31 ce
[   40.306615] RSP: 0018:ffffc9000074bcb8 EFLAGS: 00000207
[   40.307001] RAX: ffffffffa0009478 RBX: ffffc900006ad000 RCX:
   0000000000000000
[   40.307507] RDX: ffff888102d62d80 RSI: ffffc900006ad048 RDI:
   ffff888103db2c00
[   40.308026] RBP: ffffc9000074bcc0 R08: 0000000000400cc0 R09:
   ffff888103db2c00
[   40.308533] R10: ffff888105789100 R11: 0000000000000000 R12:
   0000000000000000
[   40.309049] R13: 0000000000000001 R14: ffff888105788880 R15: 00000000
   ffffffff
[   40.309558] FS:  00007f218ab76740(0000) GS:ffff888237c00000(0000) knlGS
   :0000000000000000
[   40.310138] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   40.310551] CR2: 00007f218acdcd80 CR3: 0000000103d71002 CR4:
   0000000000770ef0
[   40.311065] DR0: 0000000000000000 DR1: 0000000000000000 DR2:
   0000000000000000
[   40.311571] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7:
   0000000000000400
[   40.312085] PKRU: 55555554
[   40.312285] Call Trace:
[   40.312472]  <IRQ>
[   40.312628]  ? rcu_dump_cpu_stacks+0xe6/0x150
```

```
[   40.313435]  ? rcu_sched_clock_irq+0x528/0xff0
[   40.313766]  ? update_load_avg+0x5f/0x6a0
[   40.314057]  ? sched_slice+0x65/0x110
[   40.314324]  ? update_process_times+0x5a/0x90
[   40.314642]  ? __pfx_tick_sched_timer+0x10/0x10
[   40.314973]  ? tick_sched_handle+0x2f/0x40
[   40.315270]  ? tick_sched_timer+0x64/0x80
[   40.315559]  ? __hrtimer_run_queues+0x10a/0x2a0
[   40.315893]  ? hrtimer_interrupt+0xfb/0x230
[   40.316193]  ? __sysvec_apic_timer_interrupt+0x5b/0x130
[   40.316570]  ? sysvec_apic_timer_interrupt+0x69/0x90
[   40.316931]  </IRQ>
[   40.317091]  <TASK>
[   40.317248]  ? asm_sysvec_apic_timer_interrupt+0x1a/0x20
[   40.317632]  sk_filter_trim_cap+0xac/0x200
[   40.318851]  ? skb_copy_datagram_from_iter+0x59/0x1d0
[   40.319226]  unix_dgram_sendmsg+0x220/0xa40
[   40.319533]  ? avc_has_perm+0x8e/0x1b0
[   40.319815]  sock_write_iter+0x16d/0x180
[   40.320100]  vfs_write+0x30c/0x3a0
[   40.320351]  ksys_write+0xaa/0xe0
[   40.320595]  ? exit_to_user_mode_prepare+0x2a/0x110
[   40.320951]  do_syscall_64+0x3c/0x90
[   40.321215]  entry_SYSCALL_64_after_hwframe+0x72/0xdc
[   40.321579] RIP: 0033:0x7f218ac712c0
[   40.321879] Code: 40 00 48 8b 15 41 9b 0d 00 f7 d8 64 89 02 48 c7 c0 ff
    ff ff ff eb b7 0f 1f 00 80 3d 21 23 0e 00 00 74 17 b8 01 09
[   40.323201] RSP: 002b:00007ffc0150aff8 EFLAGS: 00000202 ORIG_RAX:
   0000000000000001
[   40.323739] RAX: ffffffffffffffda RBX: 00007ffc0150b1d8 RCX: 00007
   f218ac712c0
[   40.324247] RDX: 0000000000000040 RSI: 00007ffc0150b000 RDI:
   0000000000000005
[   40.324756] RBP: 00007ffc0150b050 R08: 0000000000000000 R09:
   0000000000000000
[   40.325265] R10: 00007f218ab924f0 R11: 0000000000000202 R12:
   0000000000000000
[   40.325778] R13: 00007ffc0150b1e8 R14: 0000558351b0add8 R15: 00007
   f218ad95020
[   40.326290]  </TASK>
QEMU: Terminated
```

C Kernel Log for Table 4

The following are the kernel logs recorded when the exploit is run without and with SafeBPF enabled.

Without SafeBPF as normal user:

```
sean@localhost:~/exp$ ./a.out
[   29.091525] SKIP bpf verifier: is_priv:  0
```

```
mapfd finished
Verifier log:
[  29.092273] BUG: unable to handle page fault for address: 00000000006d9002
[  29.092925] #PF: supervisor read access in kernel mode
[  29.093346] #PF: error_code(0x0000) - not-present page
[  29.093889] PGD 0 P4D 0
[  29.094123] Oops: 0000 [#1] PREEMPT SMP NOPTI
[  29.094570] CPU: 0 PID: 191 Comm: a.out Not tainted 6.3.8-g7a95096a8a87-dirty #8
```

Without SafeBPF as sudo:

```
sean@localhost:~/exp$ sudo ./a.out
[   126.175421] SKIP bpf verifier: is_priv:  1
mapfd finished
[  126.176212] BUG: unable to handle page fault for address: 00000000006d9002
[  126.176954] #PF: supervisor read access in kernel mode
[  126.177379] #PF: error_code(0x0000) - not-present page
[  126.177879] PGD 0 P4D 0
[  126.178153] Oops: 0000 [#1] PREEMPT SMP NOPTI
[  126.178654] CPU: 0 PID: 191 Comm: a.out Not tainted 6.3.8-g7a95096a8a87-dirty #8
```

With SafeBPF as normal user:

```
sean@localhost:~/exp$ ./a.out
mapfd finished
[  357.125244] bpf verifier: is_priv:  0
Verifier log:
bpf_prog_load finished
socketpair finished
setsockopt finished
[  357.126480] BUG: kernel NULL pointer dereference, address: 00000000000008c8
[  357.127006] #PF: supervisor read access in kernel mode
[  357.127392] #PF: error_code(0x0000) - not-present page
[  357.127776] PGD 0 P4D 0
[  357.127975] Oops: 0000 [#1] PREEMPT SMP NOPTI
[  357.128309] CPU: 1 PID: 209 Comm: a.out Not tainted 6.3.8sandbpf0.2.0 #7
[  357.128801] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.16.2-0-gea1b7a073390-p
[  357.129639] RIP: 0010:sandbox_tramp+0x57/0x140
```

With safeBPF as sudo:

```
sean@localhost:~/exp$ sudo ./a.out
[  515.814235] bpf verifier: is_priv:  1
mapfd finished
[  515.814785] BUG: kernel NULL pointer dereference, address: 00000000000008c8
[  515.815558] #PF: supervisor read access in kernel mode
[  515.816108] #PF: error_code(0x0000) - not-present page
[  515.816680] PGD 0 P4D 0
[  515.816961] Oops: 0000 [#2] PREEMPT SMP NOPTI
[  515.817426] CPU: 1 PID: 216 Comm: a.out Tainted: 6.3.8sandbpf0.2.0 #7
[  515.818491] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.16.2-0-gea1b7a073390-p
[  515.820052] RIP: 0010:sandbox_tramp+0x57/0x140
```

Here is the complete kernel log recorded during the occurrence of the page fault.

```
bpf[_ p r o2g9_.l0o9a2d2 7f3] BUG: unable to handle page fault for address
   : 00000000006d9002
[   29.092925] #PF: supervisor read access in kernel mode
9n[i s h 29e.d0
3s3o4c6k]e t#pPF: error_code(0x0000) - not-present page
[   29.093889] PGD 0 P4D 0
2i[r   f i2n9i.s0h9e4d1
2s]e Oops: 0000 [#1] PREEMPT SMP NOPTI
[   29.094570] CPU: 0 PID: 191 Comm: a.out Not tainted 6.3.8-g7a95096a8a87
   -dirty #8
ts[o c k o2p9t. 0f9i5n1i2s3h]e Hardware name: QEMU Standard PC (i440FX +
   PIIX, 1996), BIOS rel-1.16.2-0-gea1b7a073390-pr4
d
[   29.096175] RIP: 0010:sk_filter_trim_cap+0xe0/0x230
[   29.096591] Code: 5d 18 48 8b 58 18 e8 2f 1c 52 ff f6 43 02 08 0f 85 f4
    00 00 00 66 90 48 8b 43 30 48 8d 73 48 48 89 f
[   29.098105] RSP: 0018:ffffc9000077fcd0 EFLAGS: 00010246
[   29.098528] RAX: 0000000000000000 RBX: 00000000006d9000 RCX:
   0000000000000000
[   29.099071] RDX: ffff88810192db00 RSI: ffffc9000077fcbc RDI:
   0000000000000003
[   29.099645] RBP: ffff8881023ec300 R08: 0000000000400cc0 R09:
   ffff8881023ec300
[   29.100192] R10: ffff888104e29100 R11: 0000000000000000 R12:
   0000000000000000
[   29.100748] R13: 0000000000000001 R14: ffff888104e28440 R15:
   0000000000000000
[   29.101357] FS:  00007fa223476740(0000) GS:ffff888237c00000(0000) knlGS
   :0000000000000000
[   29.102053] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   29.102526] CR2: 00000000006d9002 CR3: 0000000104e8e006 CR4:
   0000000000770ef0
[   29.103146] DR0: 0000000000000000 DR1: 0000000000000000 DR2:
   0000000000000000
[   29.103713] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7:
   0000000000000400
[   29.104301] PKRU: 55555554
[   29.104545] Call Trace:
[   29.104787]  <TASK>
[   29.104988]  ? __die+0x1f/0x70
[   29.105264]  ? page_fault_oops+0x156/0x420
[   29.105602]  ? kmem_cache_alloc_node+0x49/0x240
[   29.105965]  ? exc_page_fault+0x66/0x150
[   29.106287]  ? asm_exc_page_fault+0x26/0x30
[   29.106661]  ? sk_filter_trim_cap+0xe0/0x230
[   29.107036]  ? skb_copy_datagram_from_iter+0x59/0x1d0
[   29.107426]  unix_dgram_sendmsg+0x220/0xa40
[   29.107790]  ? avc_has_perm+0x8e/0x1b0
[   29.108115]  sock_write_iter+0x16d/0x180
[   29.108445]  vfs_write+0x30c/0x3a0
```

```
[   29.108735]  ksys_write+0xaa/0xe0
[   29.108995]  ? exit_to_user_mode_prepare+0x2a/0x110
[   29.109377]  do_syscall_64+0x3c/0x90
[   29.109706]  entry_SYSCALL_64_after_hwframe+0x72/0xdc
[   29.110148] RIP: 0033:0x7fa2235712c0
[   29.110449] Code: 40 00 48 8b 15 41 9b 0d 00 f7 d8 64 89 02 48 c7 c0 ff
    ff ff ff eb b7 0f 1f 00 80 3d 21 23 0e 00 00 9
[   29.111865] RSP: 002b:00007ffdc4e49e98 EFLAGS: 00000202 ORIG_RAX:
   0000000000000001
[   29.112441] RAX: ffffffffffffffda RBX: 00007ffdc4e4a088 RCX: 00007
   fa2235712c0
[   29.113035] RDX: 0000000000000040 RSI: 00007ffdc4e49ea0 RDI:
   0000000000000005
[   29.113627] RBP: 00007ffdc4e49ef0 R08: 0000000000000000 R09:
   0000000000000000
[   29.114259] R10: 00007fa2234924f0 R11: 0000000000000202 R12:
   0000000000000000
[   29.114897] R13: 00007ffdc4e4a098 R14: 00005594b6419dd8 R15: 00007
   fa223695020
[   29.115420]  </TASK>
[   29.115590] Modules linked in:
[   29.115853] CR2: 00000000006d9002
[   29.116147] ---[ end trace 0000000000000000 ]---
[   29.116555] RIP: 0010:sk_filter_trim_cap+0xe0/0x230
[   29.116938] Code: 5d 18 48 8b 58 18 e8 2f 1c 52 ff f6 43 02 08 0f 85 f4
    00 00 00 66 90 48 8b 43 30 48 8d 73 48 48 89 f
[   29.118485] RSP: 0018:ffffc9000077fcd0 EFLAGS: 00010246
[   29.118890] RAX: 0000000000000000 RBX: 00000000006d9000 RCX:
   0000000000000000
[   29.119488] RDX: ffff88810192db00 RSI: ffffc9000077fcbc RDI:
   0000000000000003
[   29.120080] RBP: ffff8881023ec300 R08: 0000000000400cc0 R09:
   ffff8881023ec300
[   29.120638] R10: ffff888104e29100 R11: 0000000000000000 R12:
   0000000000000000
[   29.121197] R13: 0000000000000001 R14: ffff888104e28440 R15:
   0000000000000000
[   29.121720] FS:  00007fa223476740(0000) GS:ffff888237c00000(0000) knlGS
   :0000000000000000
[   29.122419] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   29.122862] CR2: 00000000006d9002 CR3: 0000000104e8e006 CR4:
   0000000000770ef0
[   29.123416] DR0: 0000000000000000 DR1: 0000000000000000 DR2:
   0000000000000000
[   29.123987] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7:
   0000000000000400
[   29.124537] PKRU: 55555554
[   29.124799] note: a.out[191] exited with irqs disabled
Killed
```