In HIS name

New algorithm for log4shell detection

Ali Sheikh Attar

Linux: eBPF Path Pruning gone wrong ***CVE-2023-2163***

Incorrect verifier pruning in BPF in Linux Kernel >=5.4 leads to unsafe code paths being incorrectly marked as safe, resulting in arbitrary read/write in kernel memory, lateral privilege escalation, and container escape..

`A flaw was found in unrestricted eBPF usage by the BPF_BTF_LOAD, leading to a possible out-of-bounds memory write in the Linux kernel's BPF subsystem due to the way a user loads BTF. This flaw allows a local user to crash or escalate their privileges on the system.`

allows an attacker to trick the eBPF verifier into thinking a register has a value different from the one it takes when executing the program.
Using this bug, an LPE exploit was written allowing the attacker to gain an arbitrary kernel memory R/W primitive that can then be used to gain a full system compromise.

Flow

1. Write program (C, BCC, bpftrace)
2. Compile (using Clang/LLVM)
3. Load (using bpf() syscall)
4. Verifier
5. Attach to hook
6. Execution

Background: What is the Verifier and What Does It Do?
To understand the bug, let's start with a brief explanation of the eBPF verifier.

The Verifier is a critical piece of the eBPF system. Before any eBPF program is allowed to run, the verifier checks it to make sure it is "safe" and won't crash the kernel or create security problems.
Problem with Verifier Path Checking

The verifier's job is hard because it needs to account for all the different ways an eBPF program could behave. For instance:

If a program has branches (like if statements or loops), the verifier must consider every possible outcome of those branches.
This can quickly become very complex because even a small number of branches can create a huge number of possible "paths" (ways the program could run).
To manage this complexity, the verifier uses a method called path pruning:

Path Pruning: The verifier tries to avoid re-checking paths that it has already determined are safe. If it sees that a particular path ends in a safe exit, it will "prune" (i.e., ignore) similar paths in the future, assuming they are also safe.

The Bug (CVE-2023-2163) in Simple Terms
Now, let's talk about the bug itself.

"Precise Tracking" Concept: When an eBPF program involves certain operations, like pointer arithmetic (calculating addresses), the verifier needs to keep a very precise track of which values (registers) are involved. This is because pointers are sensitive — mishandling them can lead to security issues.

The verifier marks these values as "precise," meaning it will closely check every possible state these values can reach.

The Flaw in Path Pruning:

The bug arises because the verifier fails to accurately track which values are "precise".
Specifically, in the example given:
There are two registers (like variables that hold data) involved: r6 and r9.
When the verifier explores a path where a pointer operation (arithmetic) involves r6, it marks r6 as "precise" and assumes it needs to check all possible states involving r6.
However, the verifier incorrectly assumes that r9 doesn't affect the "precision" of r6.
Because of this, it prematurely decides that other paths involving r6 are safe to skip or "prune."

The Consequence:

Due to this mistake, the verifier thinks all other paths (that it should have checked) are safe and skips them.
At runtime, the program takes a different path (1:2:4:6), where it uses r6 for pointer arithmetic. The verifier incorrectly assumes r6 is 0 (a safe value) when it's actually something else (unsafe), leading to a potential security issue.

In Summary: Why This Matters
The verifier's incorrect path pruning, due to failing to track r9 properly, results in missing some dangerous program behaviors.
This means an attacker could craft an eBPF program that appears safe to the verifier but actually exploits this oversight to run unsafe operations in the kernel.

Such a bug could allow an attacker to perform actions like gaining higher privileges (Local Privilege Escalation - LPE) or escaping from a container in a cloud environment.

How Was the Bug Found?
The bug was discovered using a tool called Buzzer, which generates random eBPF programs and checks for unexpected behaviors. Buzzer found a case where the verifier incorrectly pruned paths, leading to the discovery of this vulnerability.

Vulnerable linux visions https://ubuntu.com/security/CVE-2023-2163

Semon.io

Mitigation Pathways

a. Kernel Upgrade:
Certainly the most robust remedy. The latest kernel patches rectify the eBPF verifier's shortcomings, precluding the possibility of such a bytecode injection. However, a reboot is non-negotiable post this upgrade.

b. Surgical eBPF Access Curtailment:
When a reboot is logistically challenging, this approach is the paragon.

Technicals: Setting the kernel.unprivileged_bpf_disabled parameter to 1 essentially prohibits unprivileged users from leveraging eBPF system calls. This directly negates the exploit mechanism of CVE-2023-2163.

Commands for deployment:

```
echo "kernel.unprivileged_bpf_disabled=1" >> /etc/sysctl.conf
sysctl -p
```

POC

```
0: (b7) r6 = 1024
   1: (b7) r7 = 0
   2: (b7) r8 = 0
   3: (b7) r9 = -2147483648
   4: (97) r6 %= 1025
   5: (05) goto pc+0
   6: (bd) if r6 <= r9 goto pc+2
   7: (97) r6 %= 1
   8: (b7) r9 = 0
   9: (bd) if r6 <= r9 goto pc+1
  10: (b7) r6 = 0
  11: (b7) r0 = 0
  12: (63) *(u32 *)(r10 -4) = r0
  13: (18) r4 = 0xffff888103693400 // map_ptr(ks=4,vs=48)
  15: (bf) r1 = r4
  16: (bf) r2 = r10
  17: (07) r2 += -4
  18: (85) call bpf_map_lookup_elem#1
  19: (55) if r0 != 0x0 goto pc+1
  20: (95) exit
  21: (77) r6 >>= 10
  22: (27) r6 *= 8192
  23: (bf) r1 = r0
  24: (0f) r0 += r6
  25: (79) r3 = *(u64 *)(r0 +0)
  26: (7b) *(u64 *)(r1 +0) = r3
  27: (95) exit
```

Detailed Explanation of the Verifier's Process

Let's break down the entire process, focusing on where things go wrong:

1. **Initialization**:
   - Registers are initialized with specific values:
     - `r6 = 1024`, `r7 = 0`, `r8 = 0`, `r9 = -2147483648`.
2. **Modulus Operation**:
   - At **line 4**: `r6 %= 1025`.
     - The result of this operation is not precisely tracked by the verifier. So, the value of `r6` becomes **unknown**.
     - Because `r6` is now an unknown scalar, the verifier cannot determine which branch will be taken in the conditional checks (`line 6` and `line 9`).
3. **Path Exploration**:
   - The verifier begins by exploring all possible branches:
   - **First Path (False Path) - `from 19 to 21`**:
     - In this path, both `r6` and `r9` are `0`.

        - The verifier determines that the arithmetic operation `r0 += r6` is safe because adding `0` to a pointer does not change its address.
        - The verifier marks `r6` for precision tracking because it is used in a critical arithmetic operation.
        - **Backtracking** occurs here to mark earlier states of `r6` as precise.
4. **Incorrect Pruning**:
    - The verifier moves on to explore the next path (`from 9 to 11`):
        - Here, again, `r6` and `r9` are both `0` at line `19`. So, this path is also considered safe and pruned.
    - **Issue**: The verifier misses that in this path, `r6` was refined to a more precise state after line `9` (`r6` should have been known as `scalar(umin=1)`).
5. **Critical Mistake**:
    - When analyzing the path `from 6 to 9`, the verifier incorrectly prunes it, thinking that the old state (before line `6`) and the current state (after line `6`) are equivalent.
    - **State Comparison Mistake**:
        - **Old State**: `R6_rwD=Pscalar()` (precise scalar), `R9_rwD=0`.
        - **New State**: `R6_w=scalar(umax=18446744071562067968)`, `R9_w=-2147483648`.
        - The verifier mistakenly considers the two `r6` states equivalent (because the old one is a superset of the new one).
        - It does not recognize that `r9`'s values (`0` vs `0x80000000`) are different. This is because `r9` was not marked as a contributing register to the precision state of `r6`.
6. **Resulting Vulnerability**:
    - Because the verifier pruned this path incorrectly, at runtime, the program takes an unsafe path where:
        - `r6` = `0x400` and `r9` = `0x80000000` when reaching line `21`.
    - This leads to an **out-of-bounds memory access** (`r0 += r6` with a value that was incorrectly assumed to be safe).

4. Why Backtracking is Critical for Precision

- **Backtracking** helps ensure that all dependencies for a precise value are correctly tracked. If a register (`r6`) depends on another register (`r9`), and `r9` affects the precision of `r6`, then:
    - The verifier should mark both `r6` and `r9` as precise.
    - The backtracking mechanism checks all these dependencies and ensures that the precision tracking is consistent.
- **Verifier's Backtracking Mechanism**:
    - When backtracking through instructions, the verifier marks registers for precision tracking:
        - For example, if an instruction is `r6 = r7`, `r6` needs precision after this instruction, and `r7` needs precision before this instruction.

5. Summary of the Bug and Why It Happens

- The bug is due to the verifier not properly backtracking and failing to mark all necessary registers as precise.
- Because of this, it incorrectly prunes paths and considers them safe when they are not.
- The core of the problem is that the verifier does not recognize that an imprecise register (`r9`) contributes to the precision of another register (`r6`), leading to incorrect assumptions about program safety.

Final Thoughts

Understanding this bug revolves around the concept of **precision tracking** and how the verifier determines the safety of different paths in the eBPF program. The improper handling of precision tracking leads to vulnerabilities by allowing certain unsafe paths to be pruned, resulting in potential exploits such as out-of-bounds memory accesses.