

# Lessons from the buzz

What have we learned from fuzzing the eBPF  
verifier

# \$ whoami

- Software developer @ Google Montreal
- Cloud Vulnerability Research
- Into fuzzing and currently going through a Kernel hacking phase

# Agenda


- 01 Introduction
- 02 Why Buzzer?
- 03 What is Buzzer?
- 04 What have we learned so far?
- 05 Future research

# Why Buzzer?

- The eBPF verifier is complex, so is finding bugs in it
  - ~20k lines of code @ latest release
  - The verifier has a complex purpose:
    - Keep track of the state of a bpf program at each possible point (including branches)
    - Keep track of helper functions, kfuncs... etc
    - Prove that a program safe... is hard
- Other people have explored fuzzing ebpf, buzzer was inspired by Simon Scannell's blog post @ <https://scannell.io/posts/ebpf-fuzzing/>
- Provide an alternative way to play with eBPF at a "low" (i.e bytecode) level

kernel/bpf/verifier.c @ 6.11 rc7

```
21804         if (!is_priv)
21805             mutex_unlock(&priv->mutex);
21806         vfree(env->insn_aux_data);
21807     err_free_env:
21808         kfree(env);
21809         return ret;
21810     }
```



# Why Buzzer?

- But unprivileged users cannot load eBPF programs now, so why bother doing research on eBPF?
  - Attackers can still get a foothold in places with CAP\_BPF (a process, a container, etc.)
  - A secure verifier means we have a secure eBPF, paving the way for the future
  - It's fun! (and exploits are easier to write)
- What about syzkaller or other fuzzers? Why reinvent the wheel?
  - Syzkaller is amazing! We actually have plans to integrate buzzer with it
    - We aimed to look for a different set of bugs (logical bugs in verification vs memory corruption)

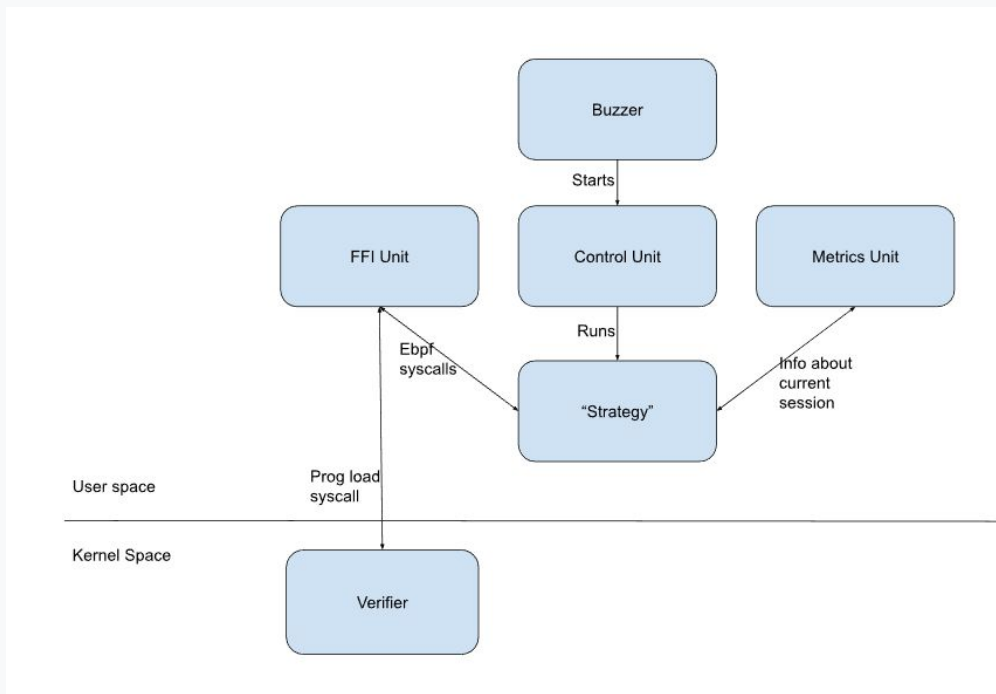
# Why Buzzer?

- A bug in the verifier means a potential path for code execution in the kernel

```
user@syzkaller:~$ whoami
user
user@syzkaller:~$ ./exploit
[.] eBPF Exploit 2024
[.] Attempting to leak a ptr to an ebf map...
[+] map_leak ffff88810544e000
[.] Loading program for arbitrary read...
[+] Program loaded!
[.] Attempting to leak map_ops...
[+] Map ops leak: ffff88810544e000
[+] ops = ffffffff8221e760
Attempting to find init_pid_ns string offset.. this will take a while, standby
Ping! looking at kernel page no 0x00, next ping at page no: 0x100
Ping! looking at kernel page no 0x100, next ping at page no: 0x200
Ping! looking at kernel page no 0x200, next ping at page no: 0x300
Ping! looking at kernel page no 0x300, next ping at page no: 0x400
Ping! looking at kernel page no 0x400, next ping at page no: 0x500
Ping! looking at kernel page no 0x500, next ping at page no: 0x600
Ping! looking at kernel page no 0x600, next ping at page no: 0x700
[+] found offset at: 68d171
[+] init_pid_ns string offset: ffffffff828ab8d1
[+] init_pid_ns address: ffffffff82a50440
[.] Attempting to find pid cred 597
[+] pid struct for process 597 is at ffff88810937aa00
[+] first at ffff8881057f05f0
[+] task_struct ffff8881057f0000
[+] process credentials at: ffff888105537780
Kernel has been pwned, standby for root shell
root@syzkaller:~# whoami
root
root@syzkaller:~# uname -a
Linux syzkaller 6.10.0-rc2-00010-g2ab795141095-dirty #30 SMP PREEMPT_DYNAMIC Tue Jun 4
2024 x86_64 GNU/Linux
root@syzkaller:~# █
```

# What is Buzzer?

- <https://github.com/google/buzzer>
- A fuzzer for the eBPF verifier that aims to:
  - Find **logical** vulnerabilities in the verifier
    - We don't focus on finding memory corruption bugs, Syzkaller does a great job at that already.
  - Provide tools to easily write eBPF programs at the bytecode level
  - Extend the research that other people have done in fuzzing ebpf (<https://scannell.io/posts/ebpf-fuzzing/>)



# What is Buzzer? - Strategies

A strategy:

- 1) Is responsible for generating eBPF programs.
- 2) Decides how to act based on verification verdicts.
- 3) Determines when a possible bug has happened

A strategy decides what type of programs to generate and how to assess the results of the verification/execution.

The rest of buzzer provides tools to interact with eBPF and visualize metrics.

```
// StrategyInterface contains all the methods that a fuzzing strategy should
// implement.
type Strategy interface {
    // GenerateProgram should return the instructions to feed the verifier.
    GenerateProgram(ffi *FFI) (*pb.Program, error)

    // OnVerifyDone process the results from the verifier. Here the strategy
    // can also tell the fuzzer to continue with execution by returning true
    // or start over and generate a new program.
    OnVerifyDone(ffi *FFI, verificationResult *fpb.ValidationResult) bool

    // OnExecuteDone should validate if the program behaved like the
    // verifier expected, if that was not the case it should return false.
    OnExecuteDone(ffi *FFI, executionResult *fpb.ExecutionResult) bool

    // OnError is used to determine if the fuzzer should continue on errors.
    // true represents continue, false represents halt.
    OnError(e error) bool

    // IsFuzzingDone if true, buzzer will break out of the main fuzzing loop
    // and return normally.
    IsFuzzingDone() bool

    // Name returns the name of the current strategy to be able
    // to select it with the command line flag.
    Name() string
}
```



# What is Buzzer? - Playground strategy

```
func (pg *Playground) GenerateProgram(ffl *units.FFI) (*pb.Program, error) {  
  
    insn, err := InstructionSequence(  
        Mov(R0, 0),  
        Exit(),  
    )  
    if err != nil {  
        return nil, err  
    }  
  
    ...  
}  
  
func (pg *Playground) OnVerifyDone(ffl *units.FFI, verificationResult *fpb.ValidationResult) bool {  
    fmt.Println(verificationResult.VerifierLog)  
    pg.isFinished = true  
    return true  
}  
  
func (cv *CoverageBased) OnVerifyDone(ffl *units.FFI, verificationResult *fpb.ValidationResult) bool {  
    ...  
    for _, addr := range verificationResult.CoverageAddress {  
        ...  
    }  
}
```

Instructions can be written in an assembly way

Strategies can have access to things like verifier log and coverage metrics

# What is Buzzer? - BTF Support

Recently thanks to the work of our Intern, Alanis Negroni, we have BTF support.

This means that we can now generate eBPF programs that are accompanied by BTF information, giving us access to a lot of new features (e.g function pointers and kfuncs)

```
types := []*btfpb.BtfType{}

// 1: Func_Proto
types = append(types, &btfpb.BtfType{
    NameOff: 0x0,
    Info: &btfpb.TypeInfo{
        Vlen: 0,
        Kind: btfpb.BtfKind_FUNCPROTO,
        KindFlag: false,
    },
    SizeOrType: 0x0,
    Extra: &btfpb.BtfType_Empty{
        Empty: &btfpb.Empty{},
    },
})

// 2: Func
types = append(types, &btfpb.BtfType{
    NameOff: 0x1,
    Info: &btfpb.TypeInfo{
        Vlen: 0,
        Kind: btfpb.BtfKind_FUNC,
        KindFlag: false,
    },
    SizeOrType: 0x01,
    Extra: &btfpb.BtfType_Empty{
        Empty: &btfpb.Empty{},
    },
})
```

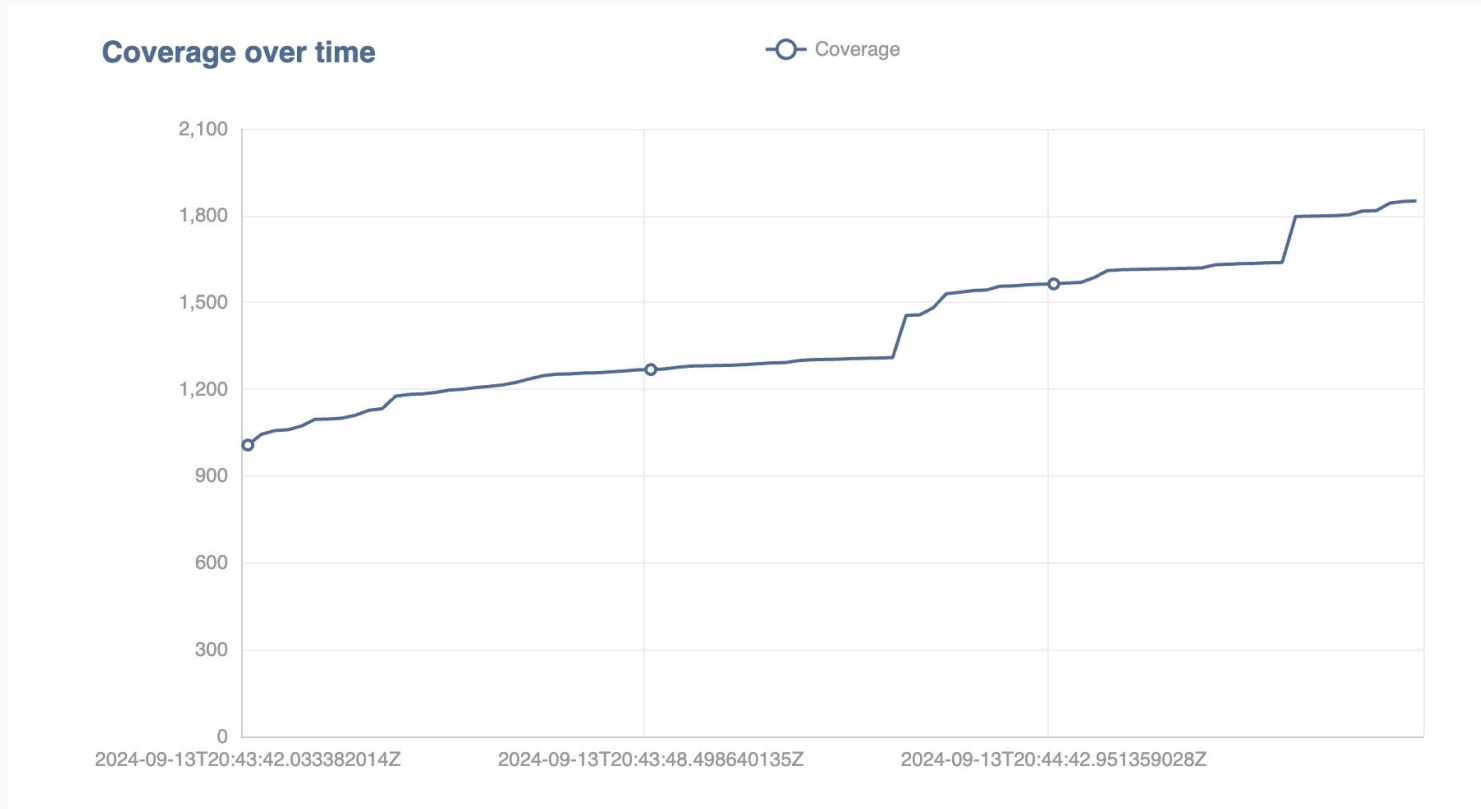
# What is Buzzer? Coverage Visualization

```

2970 static int check_subprogs(struct bpf_verifier_env *env)
2971 {
2972     int i, subprog_start, subprog_end, off, cur_subprog = 0;
2973     struct bpf_subprog_info *subprog = env->subprog_info;
2974     struct bpf_insn *insn = env->prog->insnsi;
2975     int insn_cnt = env->prog->len;
2976
2977     /* now check that all jumps are within the same subprog */
2978     subprog_start = subprog[cur_subprog].start;
2979     subprog_end = subprog[cur_subprog + 1].start;
2980     for (i = 0; i < insn_cnt; i++) {
2981         u8 code = insn[i].code;
2982
2983         if (code == (BPF_JMP | BPF_CALL) &&
2984             insn[i].src_reg == 0 &&
2985             insn[i].imm == BPF_FUNC_tail_call) {
2986             subprog[cur_subprog].has_tail_call = true;
2987             subprog[cur_subprog].tail_call_reachable = true;
2988         }
2989         if (BPF_CLASS(code) == BPF_LD &&
2990             (BPF_MODE(code) == BPF_ABS || BPF_MODE(code) == BPF_IND))
2991             subprog[cur_subprog].has_ld_abs = true;
2992         if (BPF_CLASS(code) != BPF_JMP && BPF_CLASS(code) != BPF_JMP32)
2993             goto next;
2994         if (BPF_OP(code) == BPF_EXIT || BPF_OP(code) == BPF_CALL)
2995             goto next;
2996         if (code == (BPF_JMP32 | BPF_JA))
2997             off = i + insn[i].imm + 1;
2998         else
2999             off = i + insn[i].off + 1;
3000         if (off < subprog_start || off >= subprog_end) {
3001             verbose(env, "jump out of range from insn %d to %d\n", i, off);
3002             return -EINVAL;
3003     }

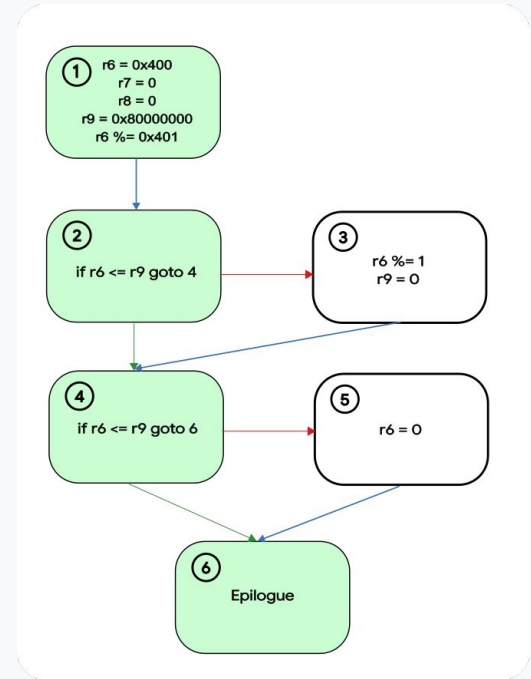
```

# What is Buzzer? Coverage Visualization



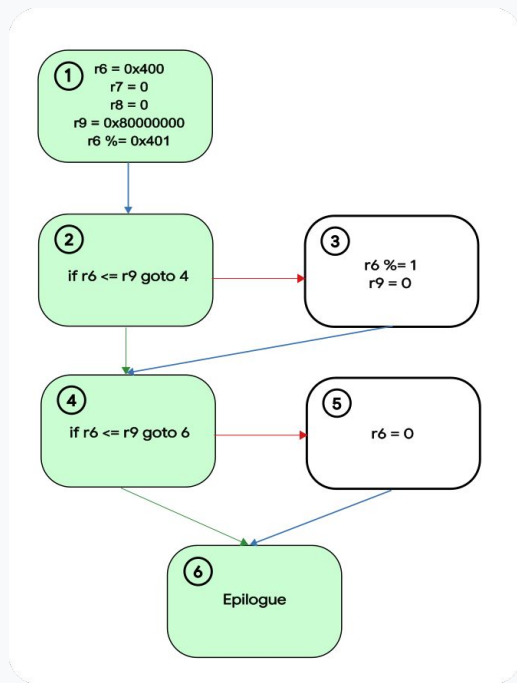
# What have we learned: CVE-2023-2163

- Bug in the verifier's branch pruning
  - Details are covered in our blog post at <https://bughunters.google.com/blog/6303226026131456/a-dee-p-dive-into-cve-2023-2163-how-we-found-and-fixed-an-ebpf-linux-kernel-vulnerability>
  - TL;DR: Buzzer found that in certain cases, the verifier would fail to mark the preciseness of some registers, leading to unsafe branches being pruned for verification, this could lead to code execution at kernel level.



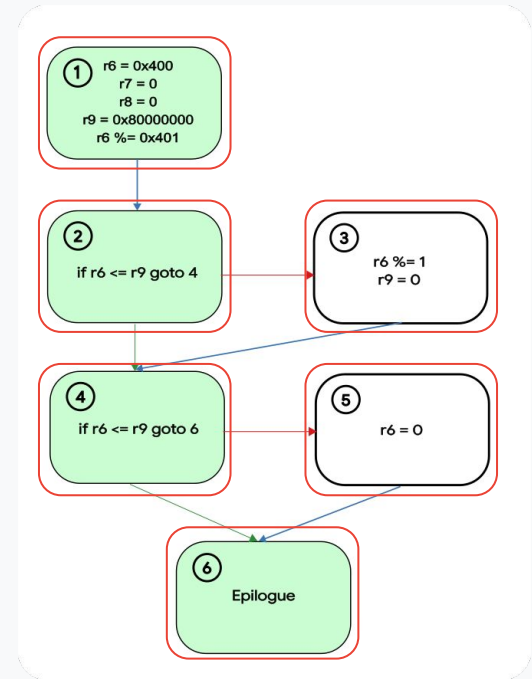
# What have we learned: CVE-2023-2163

- How was this bug found?
  - Buzzer has a strategy where it generates random jmp and alu operations
  - Then before exit it adds a register to a map pointer and tries to write to it...
  - If when we try to read that value from user space it is not there, then we know a write out of bounds might have happened



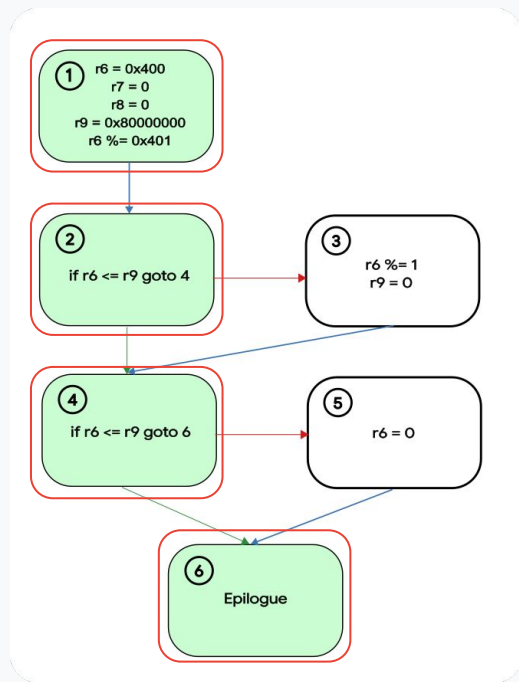
# What have we learned: CVE-2023-2163

- A bit more details on the bug
  - The verifier explores all possible branches, taking the **false** branch first
  - In the image on the right, epilogue will execute a pointer arithmetic operation with r6
  - Since R6 is set to 0, it will conclude that this path (1:2:3:4:5:6) is safe, and it will mark r6 as precise
  - However, r9 contributes to the value r6 can take (at 4) and the verifier **did not** mark it as precise too
  - At this point the verifier will mark all other branches as equivalent to 1:2:3:4:5:6 and prune them



# What have we learned: CVE-2023-2163

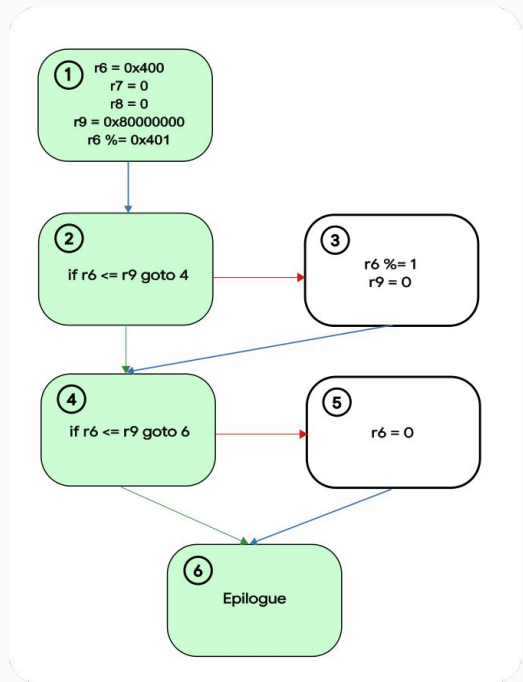
- After concluding (1:2:3:4:5:6) is safe, the verifier will prune (skip) all other paths it considers “equivalent”, in this case it is all other possible paths.
- The path that we end up taking at run time is 1:2:4:6 and since r6 is not set to 0 we can do arbitrary pointer arithmetic!
  - Again this happens because R9 was not set as contributing to the preciseness of R6, had that been the case then the verifier would not mark all other states as equivalent





# What have we learned: CVE-2023-2163

- What did we learn from this?
  - The verifier has a very complex job to do
  - This bug would have been difficult to spot via manual analysis
    - Due to the complexity of state tracking
  - Branch pruning might remain a good source for vulnerabilities, although we are yet to find another bug like this one.



# What have we learned: CVE-2024-41003

- TL;DR a bug was introduced in kernel 6.8 where it was possible to corrupt the verifier register limit tracking during branch operations
  - Details at: <https://github.com/google/security-research/security/advisories/GHSA-hfqc-63c7-rj9f>

Instruction	Verify limits assumption	Run time actual value
R1 = read_from_map()	[s32_min, s32_max]	0x7FFFFFFF
R1  = 2	[0x80000002, s32_max]	0x7FFFFFFF
If R1 != 0x7FFFFFFd (True branch)	[0x80000002, 0x7FFFFFFE]	0x7FFFFFFF
R1 -= 0x7FFFFFF0	[0x80000002, 0xE]	0xF
If R1 s>= 0xE (true branch)	[0xE, 0xE] == 0xE	0xF
R1 -= 0xE	0x0	0x1

# What have we learned: CVE-2024-41003

- Three key points to make this bug happen:
  - 1) In the program on the right it is mathematically impossible to fall through the false branch, the second bit will always be set. So R1 can never be 0x7ffffffd (d == 1101)

The verifier will nonetheless explore this false branch.

```
R1 = read_from_map() // The verifier knows nothing about R1
R1 |= 2 // The verifier knows that bit 2 is set but knows
nothing about the rest
if R1 != 0x7ffffffd goto b1:
Exit // False branch

b1:

R0 = 0 // True branch
Exit
```

# What have we learned: CVE-2024-41003

- Three key points to make this bug happen:
  - 2) When analyzing the false branch, the verifier creates a “fake” register with the constant value of the condition. Then it computes an intersect of the `var_off` of both registers and updates the value for both of them.

For this particular case, the result is

`(0x7FFFFFFF, 0x0)`

So now the verifier thinks that in the false branch, both R1 and the Fake register (constant) have a value of `s32_max`.

R1 (reg1)	Constant (reg2)
<code>(2, 0xFFFFFFFF)</code>	<code>(0x7FFFFFFF, 0)</code>

```

struct tnum tnum_intersect(struct tnum a, struct tnum b)
{
    u64 v, mu;

    v = a.value | b.value;
    mu = a.mask & b.mask;
    return TNUM(v & ~mu, mu);
}

```

```

t = tnum_intersect(tnum_subreg(reg1->var_off), tnum_subreg(reg2->var_off));
reg1->var_off = tnum_with_subreg(reg1->var_off, t);
reg2->var_off = tnum_with_subreg(reg2->var_off, t);

```

# What have we learned: CVE-2024-41003

- Three key points to make this bug happen:
  - 3) The false branch is mathematically impossible, so the program should be safe, right? The true branch will always be followed.

For the true branch, the verifier also uses a “fake” register initialized to the constant value of the condition...

But both the fake register for the false and true branch point to the same “fake” register.

So now we can influence what the verifier thinks of the true branch...

```

...
else /* BPF_SRC(insn->code) == BPF_K */ {
    err = reg_set_min_max(env,
                        &other_branch_regs[insn->dst_reg],
                        src_reg /* fake one */,
                        dst_reg, src_reg /* same fake one */,
                        opcode, is_jmp32);
}
if (err)
    return err;

```

# What have we learned: CVE-2024-41003

- Three key points to make this bug happen:
  - 3) ... And when the true branch is processed, if `s32_max_value` of the register is equal to the constant of the condition, it decreases said max by 1

```
        reg1->u32_max_value--;  
    if (reg1->s32_min_value == (s32)val)  
        reg1->s32_min_value++;  
    if (reg1->s32_max_value == (s32)val)  
        reg1->s32_max_value--;  
lse {  
    if (reg1->umin_value == (u64)val)
```

# What have we learned: CVE-2024-41003

- What did we learn from this?
  - A simple off by one in the limit tracking of the verifier is enough to write an LPE exploit!
  - The evolving nature of software opens the possibility of new bugs in well understood areas.
  - When fuzzing the verifier, monitoring the logs is also a good source of information
  - A big, yet to be fully solved, problem in buzzer is **comparing the verifier's assumptions vs the run time actual events.**

Instruction	Verify limits assumption	Run time actual value
R1 = read_from_map()	[s32_min, s32_max]	0x7FFFFFFF
R1  = 2	[0x80000002, s32_max]	0x7FFFFFFF
If R1 != 0x7FFFFFFd (True branch)	[0x80000002, 0x7FFFFFFE]	0x7FFFFFFF
R1 -= 0x7FFFFFF0	[0x80000002, 0xE]	0xF
If R1 s>= 0xE (true branch)	[0xE, 0xE] == 0xE	0xF
R1 -= 0xE	0x0	0x1

# Future research

- How to extract the verifier's assumptions of the eBPF registers and compare them with what actually goes on at runtime?
- Expand buzzer to support kfuncs and other helper functions: Alanis Negroni added support for BTF, so we can now have access to more eBPF features
- Better coverage guided fuzzing
- Fuzzing eBPF on Windows? (once support for it lands)