Alice in Kernel Land: Lessons Learned from the eBPF Rabbit Hole

Simon Scannell, Valentina Palmiotti, Juan José López Jaimez

About Us, Intro to the Speakers

Simon

- Cloud Vulnerability Researcher at Google
- Loves to travel
- Worked on a first version of an eBPF fuzzer in 2020 and found a privilege escalation in the latest Ubuntu version at the time

Valentina

- Vulnerability and Exploit Researcher at IBM X-Force Red
- Focused on low-level vulnerabilities, exploit development, and post-exploitation offensive security
- Wored on eBPF exploitation and vuln discovery concurrently to Simon, met and decided to work together in collaboration with new colleague JJ.

JJ

- Cloud Vulnerability researcher and Software Developer at Google
- Loves to bake and learn languages
- By pure coincidence: found Simon's article about the fuzzer and decided to create his own version of it… only to find out Simon was joining his team a few weeks later

eBPF - what is it?

Initially created to filter packets (classic Berkley Packet Filtering)

eBPF is a technology that allows a user mode application to run code in the kernel without needing to load a kernel module.

eBPF programs are used to do all types of things: tracing, instrumentation, hooking system calls, debugging, packet capturing/filtering, and even rootkits ;)

eBPF - why?

Developers don’t need to know how to develop kernel code

It’s easier than compiling and maintaining a custom modified kernel

It’s easy to write eBPF programs

Performance advantages to run directly in the kernel

Allows for asynchronous programming style, less context switches from user to kernel, advantageous for modern hardware.

eBPF Programs

eBPF programs are written in a high level language (C/Python)

Program is compiled into eBPF bytecode using a toolchain.

```
BPF_MOV64_IMM(BPF_REG_0, 0)
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4)
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10)
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4)
BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 1)
BPF_MOV32_IMM(BPF_REG_3, 0xFFFFFFFF)
```

eBPF: How Does it Work?

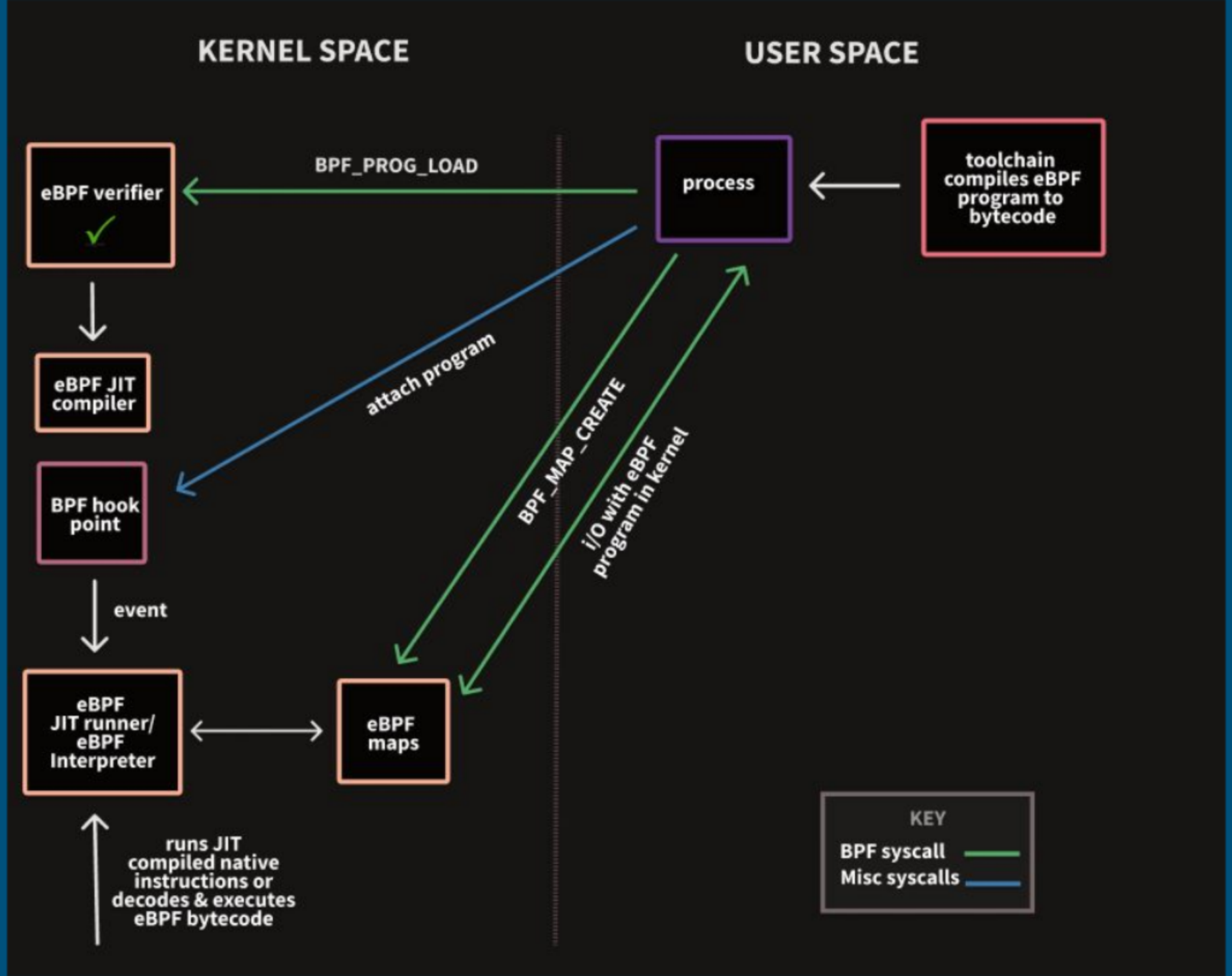
Usermode application loads byte code into kernel (via eBPF syscall)

eBPF verifier performs checks on the bytecode

If verifier checks are passed, bytecode is JIT compiled into native instruction set (or instructions are executed by interpreter while running)

Application attaches to hook point, event-based execution

Application gives input/output with eBPF maps and helper functions


KERNEL SPACE
USER SPACE
eBPF verifier
BPF_PROG_LOAD
process
toolchain compiles eBPF program to bytecode
eBPF JIT compiler
attach program
BPF hook point
BPF_MAP_CREATE
i/O with eBPF program in kernel
event
eBPF JIT runner/ eBPF Interpreter
eBPF maps
runs JIT compiled native instructions or decodes & executes eBPF bytecode
KEY
BPF syscall
Misc syscalls

eBPF Privileges

Whether unprivileged can run eBPF programs depends on sysctl knob unprivileged_bpf_disabled

Unprivileged users are limited to where programs can hook (can attach to socket the user owns)

CONFIG_BPF_UNPRIV_DEFAULT_OFF sets the sysctl knob by default. Now set by default in popular distro’s kernels like Ubuntu, beginning in 2022.

CAP_BPF Linux capability can be granted to users or containers to run eBPF programs

Fewer restrictions for programs when CAP_BPF is granted

eBPF - Applications

eBPF vulnerabilities pose a risk for applications given CAP_BPF capabilities, or running in containers with the the capability, even if unprivileged eBPF is not allowed

May see this in actual environments for:

Untrusted applications that process heavy I/O or networking activity

- Services
- Firewalls
- Security/Telemetry Applications
- Applications that run in containers, ex: Cilium

Vulnerabilities

There are 3 main places eBPF vulnerabilities can reside:

- Verifier
- JIT Compiler
- eBPF Kernel Runtime (helper functions)

When building a fuzzer, it’s important to survey the types of vulnerabilities that can occur, where and how they occur, and how to detect them in order to automate corpus generation in a way that makes sense and gain good coverage in relevant parts of the target.

Verifier Vulnerabilities - Range Calculation

Verifier keeps track of the **expected range of scalar value registers** used in an eBPF program.

Only scalars can be added to pointers to access the memory of shared maps.

Verifier has to ensure these scalar registers stay within the expected range and do not lead to out of bound access of maps.

Errors in how these ranges are calculated result in the ability to manipulate the verifier’s checks allow kernel memory corruption.

Verifier Checks Map Access

```
key := 1
value := 233
keyPtr := unsafe.Pointer(&key)
valuePtr := unsafe.Pointer(&value)
bpfmap.Update(keyPtr, valuePtr)
```

The verifier must ensure that operations like the one above, don’t result in operating on out of bounds memory

Range Calculation: CVE-2020-27194

Discovered by the first iteration of the fuzzer we're presenting today.

Caused by miscalculations of 32 bit ranges that are derived from the 64 bit registers. The patch introduced the individual tracking of 32 bit ranges for each register.

Verifier Vulnerabilities - Branching Prediction

Verifier must analyze all potential code paths of the program being loaded in order to ensure safe behavior at runtime.

Patches out instructions it believes to be in a branch that will never be reached or does not analyze branches it believes to be in the same state as already visited code paths.

Bugs in the verifier’s branching prediction mechanisms can lead to the ability to manipulate the program state to one unexpected by the verifier

Branching Prediction - CVE-2023-2163

Discovered by the second iteration of the fuzzer we’re presenting today

Verifier Vulnerabilities - General Logic Bugs

General logic bugs can occur when traversing and verifying the program. Not directly related to range/bound calculation or branching, but ultimately lead to the verifier generating an incorrect state for the program

CVE-2021-3490 -

Lead to the failure to update the bounds of a scalar registers after 32 bit AND, OR, and XOR operations. Was not an error in calculating the bounds themselves, though.

JIT Compiler Vulnerabilities - Code Generation Bugs

The eBPF JIT compiler compiles the eBPF bytecode to assembly of the native architecture of the system

JIT Compiler Vulnerabilities - Code Generation Bugs Optimization->Branch Miscalculations

CVE-2021-29154 - Issue with how branch displacements were calculated for some architectures, could lead to arbitrary shellcode execution

Highlights how architectural differences can manifest in this eBPF component

JIT Compiler Vulnerabilities - Code Generation Bugs

CVE-2021-38300 - Bug in the way classic eBPF programs were translated and compiled in the JIT compiler

Highlights that legacy cBPF program support can create issues once JIT compiled

eBPF Runtime Vulnerabilities

eBPF has “helper functions” that can interact with the eBPF program during runtime. The code for these functions reside in the “unsandboxed” area of the kernel, meaning they are just in the normal part of the kernel code base.

Vulnerabilities in the eBPF runtime can be triggered by a malicious eBPF program and be used to escalate privileges

eBPF Runtime Vulnerabilities

CVE-2021-38166 - Integer overflow and out of bounds write in hashmap lookup function

This vulnerability can be triggered if a shared eBPF map is of type hashmap and a helper function is called to retrieve a value stored in the map.

A big portion of the eBPF Runtime code attack surface is related to operating on shared maps (retrieving/storing their values, etc)

What now? Automated Vulnerability Discovery

With proper context of the target, we began our automated vulnerability discovery process by targeting the eBPF verifier via fuzzing

Automating vulnerability discovery - Why?

Verifier code is very complex (kernel/bpf/verifier.c has > 17k LoC)

Lots of changes to the source code. **CVE-2020-27194** was introduced as part of a patch for another security issue

eBPF is a standard. If we can automate vulnerability discovery, we can likely do it for more than one implementation

Automating vulnerability discovery - Fuzzing?

Pros:

- Doesn’t require a very deep technical understanding of the verifier code
- The eBPF instruction set is not very large. We can generate all opcodes
- The eBPF interface (syscalls etc.) doesn’t change much, meaning a fuzzer can be adapted relatively easy to various versions and implementations

Fuzzing: Existing solutions considered

At the time, an eBPF fuzzer for the Linux implementation existed. However, it aimed at finding memory corruption in the verifier itself

Syzkaller is a coverage-guided and sophisticated Kernel fuzzer. However:

- Built to trigger and detect memory corruption bugs. We are interested in hard-to-detect logic bugs
- Changing Syzkaller to generate, load, execute and detect errors in eBPF programs requires as much work as writing an eBPF optimized fuzzer from scratch

Fuzzing: Tailored strategy

Writing an eBPF optimized fuzzer from scratch allows us to:

- Optimize the fuzzing-input generator and tweak it for experiments
- Choose an appropriate architecture for the fuzzer
- Implement a strategy for detecting bugs

Fuzzing: Input generation

On a very high level a fuzzer just throws random stuff at a program until it crashes

The issue with this approach is that in order for our program to crash it has to pass the verifier first

That means a naive bitflipping approach likely won't generate a long list of opcodes that are:

- Correctly encoded
- Specify a valid operation
- Specify valid registers
- Specify immediates

Fuzzing: Input generation

In the first version of the fuzzer, we generated programs by:

- Prepending a static program header that performed all necessary initialization
- Generating a variable length array of opcodes
- Each opcode is a correctly encoded ALU operation taking in either a register or an immediate as an operand
- Appending a static program footer that exited the program cleanly

Fuzzing: Input generation

```
unsigned operation = 0;
// chose a random ALU operation
// for now only support operations which can leave a register in a known state
switch(rg->rand_int_range(0, 7)) {
        case 0:
                operation = BPF_ADD;
                break;
        case 1:
                operation = BPF_SUB;
                break;
        case 2:
                operation = BPF_OR;
                break;
```

Fuzzing: Input generation

```
// now, generate all the instructions
size_t index = this->header_size;
for (size_t i = 0; i < this->num_instr; i++) {

        // in most cases do an ALU operation, make sure we don't have a
        if (rg->n_out_of(8, 10) || i == this->num_instr - 1) {
                alu_instr a;
                a.generate(this->rg, this->op_reg);
```

Fuzzing: Error detection

We can now generate eBPF programs! Now what?

If a program contains pointer arithmetic and accesses **and** the verifier deems it to be valid then the verifier is confident the memory access is guaranteed to be in bounds

We can test this assumption at runtime by writing a magic value to an eBPF map during eBPF program execution

If the magic value is not in the expected position within the map, the verifier was wrong and we have a security issue!

Fuzzing: Performance and Scaling

The eBPF verifier runs with a global lock, meaning that the fuzzer won't scale well

This is bad news because we expect most of our time to be spent within the verifier. Although we have an optimized input generator, most programs will still be invalid

Fuzzing: Architecture

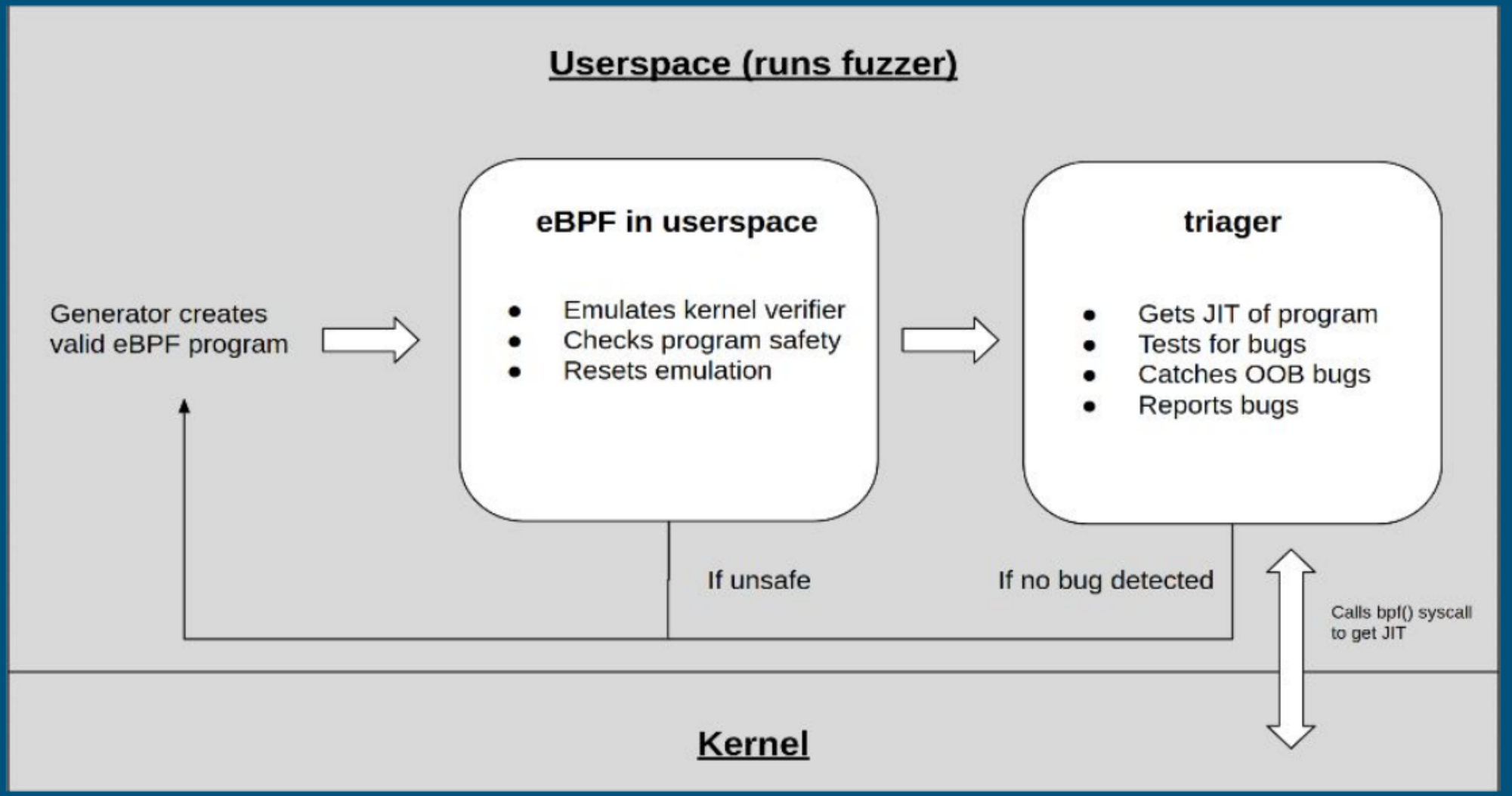
At the time, decided to extract the verifier code and compile it in user-space

This had several benefits for scalability:

- No context switches
- No verifier lock
- Easier to debug where things might go wrong

The downside: Very time consuming and difficult to port to new versions

Fuzzing: Architecture



Fuzzing: Performance and Scaling

Pros:

- Very scalable and fast due to userspace fuzzing
- Optimized eBPF input generation
- Easy to debug issues

Cons:

- Error detection is very naive
- eBPF inputs are correctly encoded when generated but still an extremely low rate for valid programs (< 1%)
- Hard to adapt to new versions

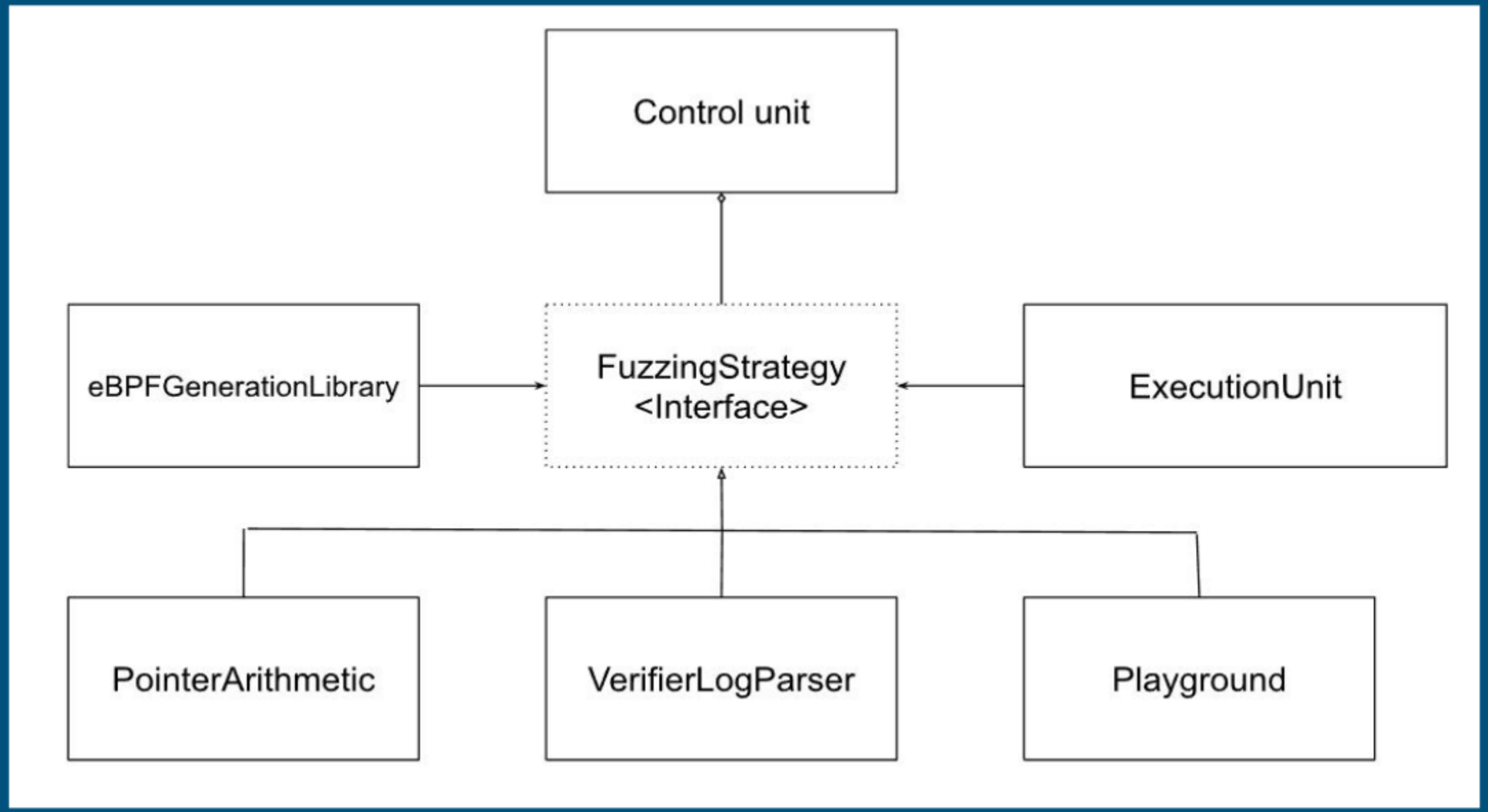
Fuzzing - A New approach

- In our previous strategy the biggest issue was that only a fraction of the generated programs were considered valid.

  What if we create a new fuzzer that generates a lot of **syntactically** valid programs

  - We look for Programs that are **semantically** wrong but the verifier gives them an O.K

High Level Architecture - Not A fuzzer, a Fuzzing framework



Fuzzing Strategies - What and Why?

- How do we detect when an error occurs?
- How do we decide how to generate a program?

Answer: Fuzing Strategies - Allow the user of the fuzzer to code their own way to generate eBPF Programs and detect when errors occur

E.G: Verifier Log Parser strategy

Verifier Log Parser strategy

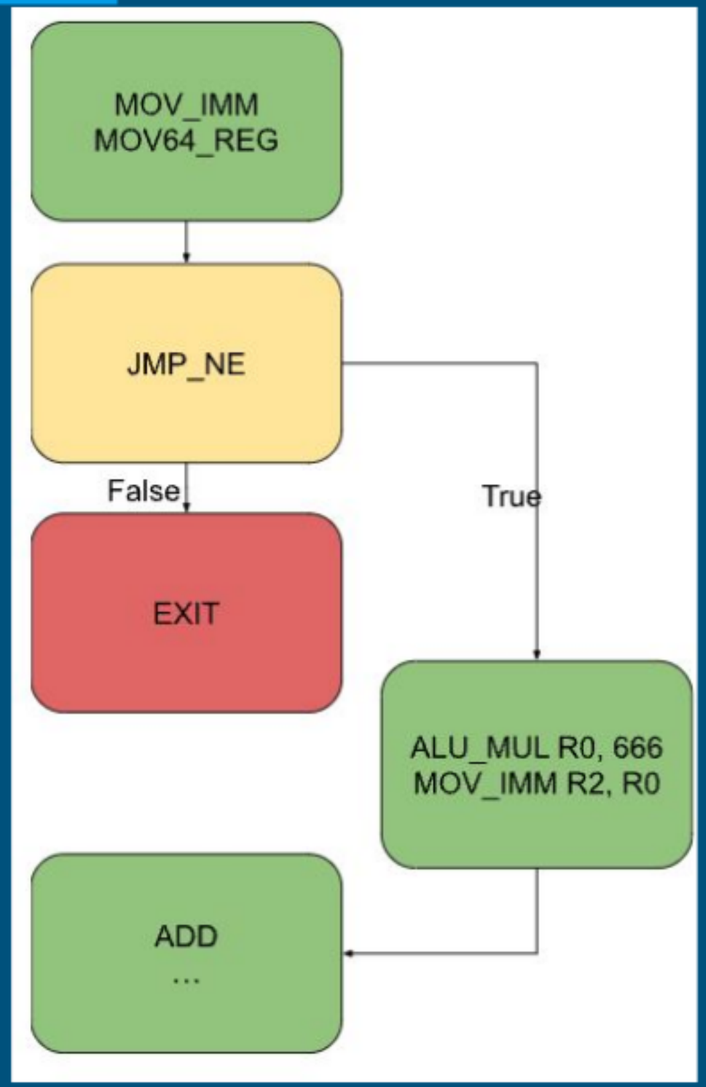
- Generation of programs:
  - Random ALU operations
- How does it detect bugs?
  - At verify time: it ingests the verifier log and parses the assumptions the verifier has made about the registers.
  - At run time: The actual values of the registers are stored in a map, then compared with the verifier assumptions

Pointer Arithmetic strategy

- Generation of programs:
  - Random ALU operations + Random JMP operations (branching)
- How does it detect bugs?
  - At the end of the program it generates a *footer* that:
    1) Chooses a random register that received ALU operations
    2) Adds the value of that register to a map pointer
    3) Attempts to write to the map
  - Then read the value from user-space, if the value is not there: OOB write happened.

This is the strategy that found CVE-2023-2163

Program Generation in a nutshell



- Programs get generated in a tree-like structure
- Once the tree generation is complete, each instruction generates its corresponding bpf bytecode (via DFS)

```
// EbpfGeneratorInterface are all the functions that a generator plugged into
// the ast is expected to have to indicate how to build the tree.
type EbpfGeneratorInterface interface {
    // Generate is the general top level function that will be invoked to
    // kick off the generation of the program.
    Generate(ast *EBPFTree) EBPFOperation

    // GenerateNextInstruction gets invoked by every instruction's GenerateNextInstruction
    // you can view it as returning the control of the construction back
    // to the generator to decide what to do next, generate more instructions
    // or finish the construction.
    GenerateNextInstruction(ast *EBPFTree) EBPFOperation
}
```

Other features - coverage information and statistics

52711 have been verified, 0 were valid (0.000000 percent were valid)

- verifier.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/kernel/bpf/verifier.c
  - covered lines: 828
- vmalloc.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/mm/vmalloc.c
  - covered lines: 220
- vsprintf.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/lib/vsprintf.c
  - covered lines: 169
- percpu.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/mm/percpu.c
  - covered lines: 91
- syscall.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/kernel/bpf/syscall.c
  - covered lines: 54
- core.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/kernel/bpf/core.c
  - covered lines: 39
- tnum.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/kernel/bpf/tnum.c
  - covered lines: 28
- disasm.c
  - full path: /usr/local/google/home/jjlopezjaimez/repos/linux-git/kernel/bpf/disasm.c

```
302         unsigned int n;
303
304         n = vscnprintf(log->kbuf, BPF_VERIFIER_TMP_LOG_SIZE, fmt, args);
305
306         WARN_ONCE(n >= BPF_VERIFIER_TMP_LOG_SIZE - 1,
307                   "verifier log line truncated - local buffer too short\n");
308
309         if (log->level == BPF_LOG_KERNEL) {
310                 bool newline = n > 0 && log->kbuf[n - 1] == '\n';
311
312                 pr_err("BPF: %s%s", log->kbuf, newline ? "" : "\n");
313                 return;
314         }
315
316         n = min(log->len_total - log->len_used - 1, n);
317         log->kbuf[n] = '\0';
318         if (!copy_to_user(log->ubuf + log->len_used, log->kbuf, n + 1))
319                 log->len_used += n;
320         else
321                 log->ubuf = NULL;
322 }
323
324 static void bpf_vlog_reset(struct bpf_verifier_log *log, u32 new_pos)
325 {
326         char zero = 0;
327
328         if (!bpf_verifier_log_needed(log))
329                 return;
330
331         log->len_used = new_pos;
332         if (put_user(zero, log->ubuf + new_pos))
333                 log->ubuf = NULL;
334 }
335
336 /* log_level controls verbosity level of eBPF verifier.
337  * bpf_verifier_log_write() is used to dump the verification trace to the log,
338  * so the user can figure out what's wrong with the program
339  */
340 __printf(2, 3) void bpf_verifier_log_write(struct bpf_verifier_env *env,
341                                            const char *fmt, ...)
342 {
343         va_list args;
344
345         if (!bpf_verifier_log_needed(&env->log))
346                 return;
```

Some statistics

It can:

- Generate ~35k eBPF Programs per minute
- Depending on the strategy, a lot of those programs might be rejected by the verifier, but that's ok. We are looking for **syntactically valid** programs that are **semantically wrong** but the verifier thinks are O.K

24 hours after we set it to run for real...

- First Strike! (CVE-2023-2163)
- It was found using the pointer arithmetic strategy
  - We added generation of random JMP operations + random ALU operations and that did the trick...
- TL;DR we discovered that the control flow analysis of the verifier is kinda broken
  - Verifier tries to simulate all possible states of branching to determine if all paths are safe
  - But the verifier is lazy... and if **some** conditions are met... then it prunes the search
  - Welp turns out that the pruning algorithm is broken, it might decide to not traverse certain paths that lead to unsafe conditions

We turned this into an LPE + Container escape

So What now?

- Code of the fuzzing framework (a.k.a *Buzzer*) has been made open source at https://github.com/google/buzzer
- There will still be some features to implement
  - Execute bpf programs across multiple vms with different OS versions
  - Better metrics collection
  - Etc.
- We just scratched the surface on how to fuzz ebpf in depth, we can find more complex vulns with more complex fuzzing strategies.

Black Hat Sound Bytes

- Granting CAP_BPF should be carefully considered as eBPF remains a complex attack surface
- When building a fuzzer, it’s important to survey the types of vulnerabilities that can occur, where and how they occur and how to detect them in order to automate corpus generation to cover relevant parts of the target
- There is a lot of value in contribution between security researchers: the lessons learned in Simon’s original fuzzer paved the way for our new Fuzzer