Unknown Title

Reliable system call interception.

Posted on 2025-01-05

Historically, intercepting Linux system calls was done with ptrace. While ptrace is more commonly known for debugging purposes, one could easily monitor system calls by using `PTRACE_SYSCALL` (or even `PTRACE_SYSEMU`) to wait for the traced process to make a system call, then send off `PTRACE_GETREGS` and `PTRACE_SETREGS` to read and write the registers associated with the system call.

So while the Linux kernel always had the facilities to monitor, fake, modify and restrict system calls, the glaring problem with ptrace is that it is very slow, as it stops twice for every system call (unless `PTRACE_SYSEMU` is used) and there is no way to natively filter for a specific set of system calls. It gets worse, because reading and writing to the registers is incredibly cumbersome and one quickly encounters architecture-specific quirks.

This is where seccomp user notify comes in, where recent advancements by Christian Brauner have made it possible to intercept system calls in a much more elegant way. Due to the addition of BPF it can be programmed to yield back only for the desired system calls, which significantly reduces the performance penalty and unaffected sections of the traced program run almost as if no tracer was attached at all. This is also similar to what strace is doing with the --seccomp-bpf option as a means for lessening the performance overhead, although it still uses ptrace for the main functionality.

Usecase

A few years ago, I wrote a tool called copycat that uses this mechanism to dynamically intercept all `open()`-style system calls made by a supervised process and returns, depending on some rules, either the requested file or a completely different faked file. This can be very useful in some situations, for example when a program is hardcoded to use one specific location for a configuration file, but you rather want to use a different location.

The replacement of the opened file is completely transparent to the application and can easily be configured with simple environment variables. For example, the following snippet will trick `cat` into outputting `/tmp/b` instead of `/tmp/a`:

```
COPYCAT="/tmp/a /tmp/b" copycat -- cat /tmp/a
```

What happens behind the curtains, actually involves a lot more detail than just intercepting system calls. For one, it is also necessary to inject file descriptors directly into the file descriptor table of the traced process, as otherwise the faked file would only be valid in the tracer process.

seccomp unotify

Originally seccomp user notify was intended for container usecases, but we can use it just as easily for normal processes by adopting the age-old fork+exec pattern. The child process simply registers a seccomp filter with SECCOMP_SET_MODE_FILTER and then executes the target application, while the parent process acts as the supervisor and repeats an ioctl loop with the special SECCOMP_IOCTL_NOTIF_RECV flag, which will yield any time the supervised process attempts a matching system call.

A few special prerequisites have to be met to make this work. First the child process needs to drop all privileges.

```
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
```

This is necessary, as otherwise an unprivileged process could execve a setuid program with a malicious filter attached.

> Such a malicious filter might, for example, cause an attempt to use setuid to set the caller's user IDs to nonzero values to instead return 0 without actually making the system call.

On the supervisor side we have to make some extra checks due to a kernel bug that does not notify the supervisor when the supervised process exits. Since Linux 6.11 the bug is resolved, so in that case the `SECCOMP_IOCTL_NOTIF_RECV` loop is sufficient and will return with `ENOENT`, when the child terminates. However on older kernel versions that `ioctl` would hang forever, so an easy workaround is to install a signal handler for `SIGCHLD` with sigaction. Just keep in mind to do the old Unix dance of just using async-signal-safe functions inside of it, in particular no allocations or locks. Alternatively it is possible to epoll the file descriptor returned when registering the BPF filter.

Finally when the supervisor handles an intercepted system call received by `SECCOMP_IOCTL_NOTIF_RECV`, the `struct seccomp_notif *req` contains all the system call's arguments as part of its `data.args` array. Except that is not the whole truth, because while arguments fitting into one register are usually directly visible there, larger arguments (such as the file name to open) are passed as a pointer. Thus, all information that we then get at this stage is a useless pointer pointing into the memory of another process.

```
long syscall(SYS_open, const char *pathname, int flags, mode_t mode)
```

So we end up having to open `/proc/$PID/mem` just to read the `pathname`. Luckily we do not get any problems with yama security policies, as the seccomp operations already require us to have a predefined relationship between the supervisor and supervised process anyway, which in this case means one is the parent of the other. If you feel this is hacky, wait until you think about all the TOCTOU opportunities when we read the memory referred to by just a PID. Here seccomp can help us out: As long as we read it before we continue the syscall **and** confirm the notification ID is still valid with SECCOMP_IOCTL_NOTIF_ID_VALID, we are safe.

Now that we have all the system call arguments, we can decide if we want to allow it or modify it and return it with a different file. In both cases we need to use the `struct seccomp_notif_resp *resp` parameter . If we want to allow the system call normally, we can just set its `flags` field to SECCOMP_USER_NOTIF_FLAG_CONTINUE and send back the response with SECCOMP_IOCTL_NOTIF_SEND.

```
resp->flags |= SECCOMP_USER_NOTIF_FLAG_CONTINUE;
resp->error = 0;
resp->val = 0;
ret = ioctl(listener, SECCOMP_IOCTL_NOTIF_SEND, resp)
```

However, it becomes slightly more involved if we want to modify the system call arguments. In that case we need to open the faked file on the supervisor side, pretend like the originally intended system call worked and return the file descriptor number of the faked file to the traced process. Except there is the huge problem that the file descriptor will obviously not be valid in the target process.

> The curious reader might wonder why we do not just rewrite the system call arguments to the faked file, and then let the process continue the system call normally. Again, the argument points to memory inside the traced process. Rewriting memory is not transparent to the process, but injecting file descriptors is.

This is where SECCOMP_IOCTL_NOTIF_ADDFD comes in. It will atomically both install a file descriptor directly into the file descriptor table of the target process and return it as part of the system call.

```
struct seccomp_notif_addfd addfd = {};
addfd.id = req->id;
addfd.flags = SECCOMP_ADDFD_FLAG_SEND;
addfd.srcfd = ret;
resp->error = 0;
resp->val = ret
ret = ioctl(listener, SECCOMP_IOCTL_NOTIF_ADDFD, &addfd);
close(addfd.srcfd);
```

After we have injected the file descriptor, we can simply close it on our side.

BPF filter

For the most part now we have overglossed the most important detail, which is the BPF filter that decides if we want to intercept a system call. We can always continue a system call normally from our handler with

SECCOMP_USER_NOTIF_FLAG_CONTINUE, but a BPF filter is crucial for skipping this expensive roundtrip in the first place.

While eBPF has made some fame in the tracing and profiling scene lately, seccomp uses the original unextended Berkeley packet filter. Both instruction sets are quite similar and the Linux kernel actually internally translates BPF to an eBPF representation. Since BPF filters run in kernel space, static checks are done to make sure that they do not crash and that they terminate. There is no difficulty in solving the halting problem for circle-free programs, so the eBPF verifier uses a simple DFS to check that, i.e. loops are not allowed. For most architectures, the kernel can also JIT compile eBPF to native machine code.

Essentially the BPF instruction set has two registers, A and X, but the kernel C definitions refer to them as BPF_K and BPF_X. We can load into these registers with the BPF_LD instruction (e.g. 32-bit wide with BPF_W) and BPF_JMP instructions allow us to jump based on comparing a register value with a given value. For example BPF_JUMP(BPF_JMP+BPF_JGE+BPF_X, 42, jt, jf) will increase the instruction pointer by jt, if the value in the X register is greater or equal to 42. Otherwise it will increase it by jf. The instruction pointer will also further be increased by one after each instruction.

With the BPF_RET instruction we can finally return a value, which the kernel then will use to decide what to do with the system call. So a BPF filter to intercept a certain set of system calls nrs of length len would look a little something like this:

```
int trap_syscalls(const int *nrs, size_t len, unsigned int flags) {
        struct sock_filter filter[MAX_FILTER_SIZE];
        int i = 0;

        filter[i++] = BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct
seccomp_data, arch));
        filter[i++] = BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 0,
2);
        filter[i++] = BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct
seccomp_data, nr));
        filter[i++] = BPF_JUMP(BPF_JMP+BPF_JGE+BPF_K, X32_SYSCALL_BIT, 0, 1);
        filter[i++] = BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL_PROCESS);

        for (int j = 0; j < len; ++j) {
                filter[i++] = BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, nrs[j], len -
j, 0);
        }

        filter[i++] = BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW);
        filter[i++] = BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_USER_NOTIF);
        struct sock_fprog prog = {
                .len = (unsigned short) i,
                .filter = filter,
        };
        return seccomp(SECCOMP_SET_MODE_FILTER, flags, &prog);
}
```

There is a lot to unpack here, so let's go through the individual BPF instructions one by one.

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct seccomp_data, arch))
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 0, 2)
```

The first two instructions load the architecture and check that it matches our expectations. To understand why this is important, we can refer directly to the common pitfalls section in the official documentation:

> On any architecture that supports multiple system call invocation conventions, the system call numbers may vary based on the specific invocation. If the numbers in the different calling conventions overlap, then checks in the filters may be abused. Always check the arch value!

The next two instructions check for something similar:

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct seccomp_data, nr))
BPF_JUMP(BPF_JMP+BPF_JGE+BPF_K, X32_SYSCALL_BIT, 0, 1)
```

The `arch` field is actually not unique for all calling conventions. For example, both the x86-64 ABI and the x32 ABI use `AUDIT_ARCH_X86_64`, so the only way to tell them apart is by checking if the `__X32_SYSCALL_BIT` is set. Furthermore, if system calls are denied only based on its exact `nr`, then a malicious program could simply set `__X32_SYSCALL_BIT` to bypass this filter.

If any of these checks fail, the jump location is the following BPF instruction, which results in immediate termination of the process that made the system call.

```
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL_PROCESS)
```

Now with all the fun boilerplate done, we can finally insert BPF instructions for all passed system call numbers, that check if we want to intercept or just pass-through that specific system call.

```
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, nrs[j], len - j, 0)
```

This jump to the intercept instruction is a rats nest of off-by-one errors: The `jump-true` branch is actually more like `(len - 1) - j + 2 - 1`. The intercept instruction is the second instruction after the for loop, so we have to jump to the end of the for loop (which has index `len - 1`) relatively from the current index `j`, then jump to the second instruction, but subtract one again, because BPF automatically increments the instruction pointer by one after each instruction.

Then the final instructions are the jump targets from all the previous checks.

```
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW)
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_USER_NOTIF)
```

The first one is reached if none of the checks in the for loop match and simply allows the system call normally. The second one kicks off interception of the system call and will return back to our userspace handler that waits in an `ioctl()` loop with the `SECCOMP_IOCTL_NOTIF_RECV` flag. For installing the BPF filter we simply use SECCOMP_SET_MODE_FILTER at the end. For a more complete picture have a look at the source code yourself, specifically seccomp_exec.c for handling the system calls and seccomp_trap.c for registering the BPF filter. There is also a smaller sample in the Linux kernel source tree to get started.

Finally it should be emphasized that seccomp unotify should never be used to implement security policy decisions. The TOCTOU attacks alone hidden here make this impossible, for example if the supervisor signals `SECCOMP_USER_NOTIF_FLAG_CONTINUE`, the system call will in fact continue, but the process still has a small opportunity window to rewrite the system call arguments before it actually runs. However, it is still a great tool to intercept system calls with minimal performance impact.