Module: Sandboxing

Escaping seccomp

Yan Shoshitaishvili
Arizona State University

Breaking out

Generally, to do anything useful, a sandboxed process needs to be able to communicate with the privileged process.

Normally, this means allowing the sandboxed process to use *some* system calls. This opens up some attack vectors:

- permissive policies
- syscall confusion
- kernel vulnerabilities in the syscall handlers

Permissive Policies

Combination of:

1. System calls are complex, and there are a lot of them...
2. Developers might avoid breaking functionality by erring on the side of permissiveness.

Well-known example: depending on system configuration, allowing the `ptrace()` system call could let a sandboxed process to "puppet" a non-sandboxed process.

Some less well-known effects:

- `sendmsg()` can transfer file descriptors between processes
- `prctl()` has bizarre possible effects
- `process_vm_writev()` allows direct access to other process' memory

Syscall Confusion

Many 64-bit architectures are *backwards compatible* with their 32-bit ancestors:

```
amd64 / x86_64 ¦ x86
       aarch64 ¦ arm
        mips64 ¦ mips
     powerpc64 ¦ ppc
       sparc64 ¦ sparc
```

On some systems (including amd64), you can switch between 32-bit mode and 64-bit mode *in the same process*, so the kernel must be ready for either.

Interestingly, system call numbers differ between architectures, including 32-bit and 64-bit variants of the same architecture!

Policies that allow both 32-bit and 64-bit system calls can fail to properly sandbox one or the other mode.

Example: `exit()` is syscall 60 (`mov rax, 60; syscall`) on amd64, 1 (`mov eax, 1; int 0x80`) on x86.

Kernel Vulnerabilities

The last resort...

If the seccomp sandbox is correctly configured, the attacker can't do anything useful...

But they can still interact with the system calls that *are* allowed! This allows the attacker to try to trigger vulnerabilities in the kernel.

Powerful! Over 30 Chrome sandbox escapes in 2019 alone: https://github.com/allpaca/chrome-sbx-db

Stay tuned for the Kernel Exploitation module!

Unless....?

Think: what is your goal, as an attacker?

Is it always code execution?

Screaming into the void...

Often, your goal is data exfiltration (like /flag!).

Even if you can't directly communicate with the outside world, often you can send "smoke signals":

- Runtime of a process (see `sleep(x)` system call) can convey a lot of data.
- Clean termination or a crash? This can convey one bit.
- Return value of a program (`exit(x)`) can convey one byte.

Real-world example: attackers use DNS queries to bypass network egress filters.

As long as you can communicate **1 bit**, you can repeat the attack to get more and more bits!