LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Extensible Syscalls

Christian Brauner (Canonical)
<`christian.brauner@ubuntu.com`>

Aleksa Sarai (SUSE)
<`cyphar@cyphar.com`>

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Groundhog Day

- We need an agreement on how we design future APIs.
  - Same argument with each new API proposal.
  - Either people want ...
    1. to have a full-blown multiplexer that is so generic that ideally it spans multiple APIs at once; or
    2. individual syscalls for each operation.
  - We've burned userspace and ourselves with both of them already.
  - There are no good guidelines to follow for developers.

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Groundhog Day

- We should have *documented* stronger requirements for newer syscalls.
  - A baseline of extensibility for new syscalls (to avoid the `renameat2/readlinkat/dup/accept/`… problem).
  - Note that we're not proposing that using structs in syscalls is the new baseline for extensibility. The baseline of extensibility should be the requirement of a flag argument.
    - Arguably this is the case today, though there are quite a few syscalls without a flag argument that should have one…

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Alternative ways of dealing with extensions

1. Always require a new syscall.
   - Costly for userspace to adapt their codebases to new syscalls all the time. This is especially true for shared libraries (doubly so with seccomp).
   - That's the other extreme of having full-blown `ioctl`-style multiplexers.
   - 64-bit flag arguments.
2. Add a new flag and have a fun time with `va_arg` (`fcntl`-style).
   - Glibc has had "fun" with this (`O_TMPFILE`), and it's pretty damn ugly.
   - *64-bit flag arguments.*
3. Fixed-size struct with "enough" padding.
   - We are bad at predicting the future.
   - Why not go the extra step and not worry about having good predictions?
4. Buy a time machine and skip extensions.
   - Can I have one too, please?
   - 128-bit flag arguments (probably).

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Extensible structs - a modest compromise

- Introduction of new syscalls is a pain for userspace.
  - There should be a compromise between having a dedicated syscall for everything and stuffing everything into a single syscall.
- Extensible structs are a modest proposal one step further from flag-based extensibility but far away from `ioctl` madness:

`creat(2)` -- `open(2)` -- `openat2(2)` -- `prctl(2)` -- `ioctl(2)`

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Why add more fuel to the fire?

- Support APIs that can be expected to grow beyond just adding flags.
- Need a better compromise between maintainability and simplicity.
- Alternatives are inconvenient (there are better uses for time travel).
- Better to deal with this fire here and now rather than having the same (derailing) argument on every patchset, which discourages new contributors and leads to inconsistent APIs.
    - And we can document the conclusions of this discussion to avoid bringing this issue up again.

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Extensible structs

```
int openat2(int fd, const char *path,
            struct open_how *how, size_t usize);
```

- New fields always appended.
  - Zero value in new fields means "old behavior"
- **(ksize == usize)** Copy the struct verbatim.
- **(ksize > usize)** Copy usize bytes, zero-fill trailing bytes.
- **(ksize < usize)** Copy ksize bytes, check if trailing bytes are zeroed. If non-zero bytes are present, return `-E2BIG`.
- Dedicated kernel helper called copy_struct_from_user().

```
struct open_how how = {
        .flags      = O_RDONLY,
        .resolve    = RESOLVE_IN_ROOT,
        .newfield   = SOME_NEW_FLAG,
};

int fd = openat2(dfd, path,
                      &how, sizeof(how));
/* -E2BIG if .newfield not supported. */
/* -EINVAL if O_RDONLY not supported. */
if (fd < 0)
        return -1;
```

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

No more multiplexers

- Extensible structs are _not_ multiplexers. They especially need to be discerned from multiplexers with polymorph types passed through a void pointer or a union or a long...
  - For example, `bpf(2)` is a multiplexer making use of extensible structs not an extensible struct based syscall.
  - Most people don’t want to introduce any new polymorph multiplexers. That seems almost universally agreed upon.
  - glibc is explicitly advising against the introduction of new multiplexers as they are hard to deal with in shared libraries.

va_arg … more like bad_arg

- `va_arg` style extension (as used by multiplexers like `fcntl(2)` and `prctl(2)`) deserve explicit mention:
  - Ugly to use due to the need to indicate which arguments should be ignored (either through flags or command enum).
  - Are limited to 5-6 arguments depending on architecture.
  - Glibc has to call `va_arg(3)` unconditionally because they don’t know if a future command will require more arguments (the upshot is that they pass garbage from the stack and hope the kernel ignores it and there’s enough data in the stack).

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

The “cr*p insertion vector” argument

- Someone raised the concern that extensible structs can be abused to “sneak in” problematic features without sufficient review, while being forced to introduce a new syscall in order to support the same new feature would cause a more thorough review.

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

The “cr*p insertion vector” argument

- There are at least four counter-points:
    1. For a start, the features that prompted that reaction were all caught during review so there’s doubt that this is even a real problem. Especially for high-profile subsystems such as the vfs and core kernel.

    2. To the extent that this is a problem, it is no more of a problem than with flags and other extension designs -- subsystem maintainers should already be reviewing ABI changes enough to avoid this. All this design adds is the ability to easily add new fields to structs…

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

The “cr*p insertion vector” argument

3. It is debatable that syscalls prompt a more thorough review. We have a bunch of syscalls with very questionable APIs (`keyctl(2)`, `dup(2)`, etc.). [There’s even an LWN article about it.]()

4. All of the existing “questionable features” proposed could be applied to flags (`O_MAYEXEC` was originally an `openat(2)` patch). Is the ability to easily add new fields to existing struct arguments really the only thing stopping questionable features from being snuck in?

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Checking for supported features

- Userspace needs to know what features are supported in a given syscall. This is usually done by having a (painful) trial and error approach.
    - Userspace has to write elaborate cosplay scenarios where we have to exercise the feature to see if it works without borking the system.
    - Emulation makes this worse because emulated syscalls act strange.
    - This is maybe fine for long-running system daemons but it certainly isn’t workable for shared libraries or shorter-running programs (nor for our sanity -- see the LXD code for this).
- It should be possible to do this in a much simpler way -- and thanks to the design of extensible struct argument syscalls we can do it in a manner which is backward- and forward-compatible.

Checking for supported features

- Here's an initial proposal.
- Already discussed on libc-alpha mailing list in the context of clone3(2) and [more broadly at Linux.conf.au 2020](#).
- Syscall is **no-op** (returns some errno) and "returns" version of struct where:
    - All valid flag bits in flag fields are set.
    - All non-flag fields are filled with 0xFF. (Could be implemented as way of describing limits for fields.)
- Userspace then looks at the final struct to determine feature support.

```
struct clone_args arg = {
         // (1 << 63)
        .flags = SUPPORTED_BITS
};
int ret = clone3(&arg, sizeof(arg));
assert(ret < 0); // always fails
if (errno != E_SUPPORTED_BITS_NOOP) {
        // kernel without feature check
        if (errno == EINVAL)
                args = CLONE3_INIT_VERSION;
        else
                return -1;
}

bool abc_supported = args.flags & CLONE_ABC;
bool xyz_supported = args.xyz != 0;
```

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Checking for supported features

- **(ksize == usize)** Copy the struct verbatim.
- **(ksize > usize)** Copy usize bytes, any trailing bytes on the kernel side are **ignored**.
- **(ksize < usize)** Copy ksize bytes, zero-fill the trailing (usize - ksize) bytes (unknown extensions have their zero value defined as the “old behaviour”).
- New copy_struct_to_user() helper.

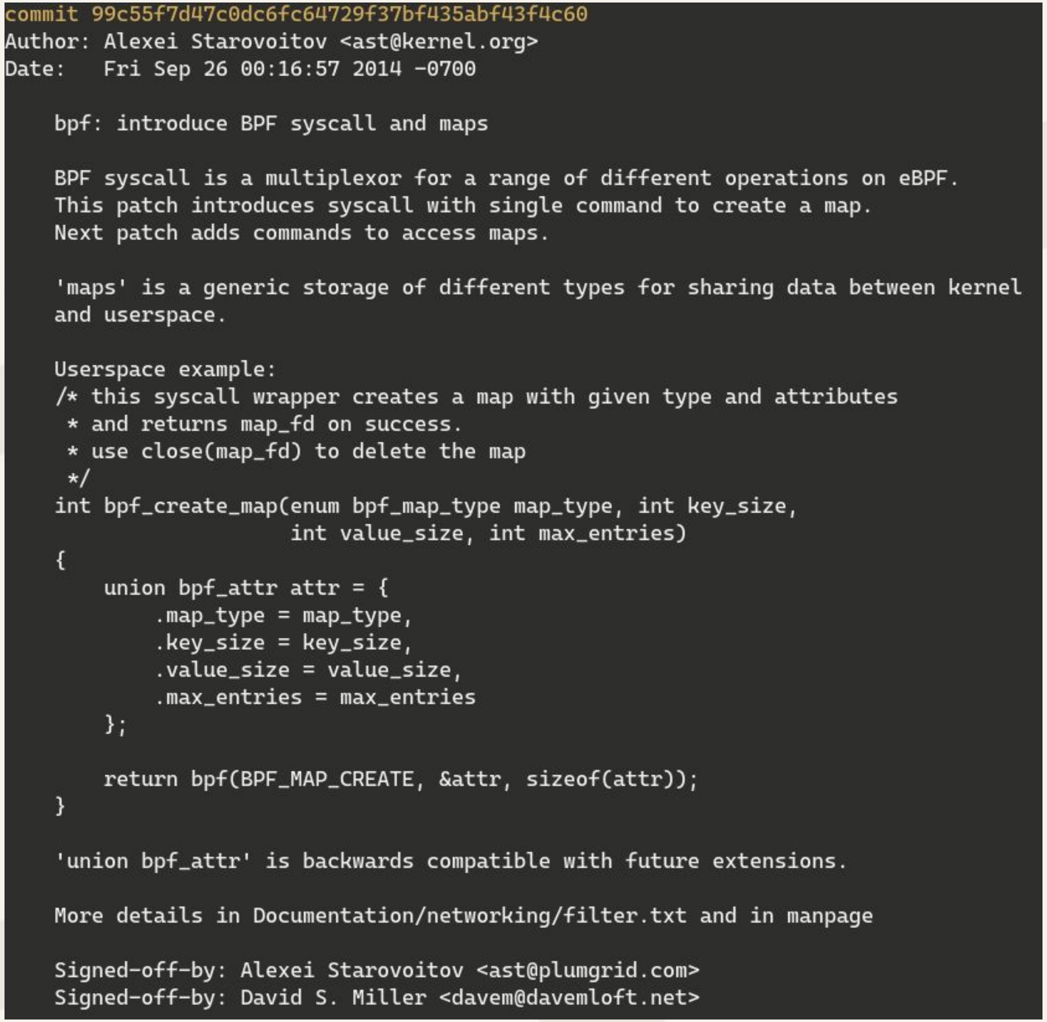
```
struct clone_args arg = {
       // (1 << 63)
      .flags = SUPPORTED_BITS
};
int ret = clone3(&arg, sizeof(arg));
assert(ret < 0); // always fails
if (errno != E_SUPPORTED_BITS_NOOP) {
      // kernel without feature check
      if (errno == EINVAL)
            args = CLONE3_INIT_VERSION;
      else
            return -1;
}

bool abc_supported = args.flags & CLONE_ABC;
bool xyz_supported = args.xyz != 0;
```

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Making this part of the kernel docs

- Documentation/process/adding-syscalls.rst is of questionable currency.
  - Our first attempt to update it went largely unnoticed.
- Proposal to update adding-syscalls.rst:
  - Formalise how extensible structs are intended to be used.
    - `perf_event_open` is currently mentioned
    - … but `clone3`/`openat2` are more modern.
  - Mention new variable argument or type polymorph multiplexers are not encouraged.
  - Mention syscalls with a flag argument for extensibility should use `unsigned int`.
  - Describe common issues the glibc folks have (such as pointer types in structs, et al) as well as “C annoyances” such as explicit padding and aligned fields.



```
commit 99c55f7d47c0dc6fc64729f37bf435abf43f4c60
Author: Alexei Starovoitov <ast@kernel.org>
Date:   Fri Sep 26 00:16:57 2014 -0700

    bpf: introduce BPF syscall and maps

    BPF syscall is a multiplexor for a range of different operations on eBPF.
    This patch introduces syscall with single command to create a map.
    Next patch adds commands to access maps.

    'maps' is a generic storage of different types for sharing data between kernel
    and userspace.

    Userspace example:
    /* this syscall wrapper creates a map with given type and attributes
     * and returns map_fd on success.
     * use close(map_fd) to delete the map
     */
    int bpf_create_map(enum bpf_map_type map_type, int key_size,
                       int value_size, int max_entries)
    {
        union bpf_attr attr = {
            .map_type = map_type,
            .key_size = key_size,
            .value_size = value_size,
            .max_entries = max_entries
        };

        return bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
    }

    'union bpf_attr' is backwards compatible with future extensions.

    More details in Documentation/networking/filter.txt and in manpage

    Signed-off-by: Alexei Starovoitov <ast@plumgrid.com>
    Signed-off-by: David S. Miller <davem@davemloft.net>
```

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

va_arg … more like really_bad_arg

- `va_arg` style multiplexers cause even more issues for glibc in general:
  - Different argument types cause confusion.
  - `size_t` and `long` cannot be differentiated and this causes issues on x32-compat due to differences between the kernel and C calling conventions (kernel requires zero or sign extension).
  - Changing a type of an argument for a single variant requires us to port all users of the syscall to a new version (see `futex(2)` and 64-bit `time_t` -- all `futex` users needed to be ported even though only a few `futex` operations actually use `time_t`).