user_events: User-based Event Tracing

Author: Beau Belgrave

Overview

User based trace events allow user processes to create events and trace data that can be viewed via existing tools, such as ftrace and perf. To enable this feature, build your kernel with CONFIG_USER_EVENTS=y.

Programs can view status of the events via /sys/kernel/debug/tracing/user_events_status and can both register and write data out via /sys/kernel/debug/tracing/user_events_data.

Programs can also use /sys/kernel/debug/tracing/dynamic_events to register and delete user based events via the u: prefix. The format of the command to dynamic_events is the same as the ioctl with the u: prefix applied.

Typically programs will register a set of events that they wish to expose to tools that can read trace_events (such as ftrace and perf). The registration process gives back two ints to the program for each event. The first int is the status index. This index describes which byte in the /sys/kernel/debug/tracing/user_events_status file represents this event. The second int is the write index. This index describes the data when a write() or writev() is called on the /sys/kernel/debug/tracing/user_events_data file.

The structures referenced in this document are contained with the /include/uap/linux/user_events.h file in the source tree.

NOTE: *Both user_events_status and user_events_data are under the tracefs filesystem and may be mounted at different paths than above.*

Registering

Registering within a user process is done via ioctl() out to the /sys/kernel/debug/tracing/user_events_data file. The command to issue is DIAG_IOCSREG.

This command takes a struct user_reg as an argument:

```
struct user_reg {
      u32 size;
      u64 name_args;
      u32 status_index;
      u32 write_index;
};
```

The struct user_reg requires two inputs, the first is the size of the structure to ensure forward and backward compatibility. The second is the command string to issue for registering. Upon success two outputs are set, the status index and the write index.

User based events show up under tracefs like any other event under the subsystem named "user_events". This means tools that wish to attach to the events need to use /sys/kernel/debug/tracing/events/user_events/[name]/enable or perf record -e user_events:[name] when attaching/recording.

NOTE: *The write_index returned is only valid for the FD that was used*

Command Format

The command string format is as follows:

```
name[:FLAG1[,FLAG2...]] [Field1[;Field2...]]
```

Supported Flags

None yet

Field Format

```
type name [size]
```

Basic types are supported (__data_loc, u32, u64, int, char, char[20], etc). User programs are encouraged to use clearly sized types like u32.

NOTE: *Long is not supported since size can vary between user and kernel.*

The size is only valid for types that start with a struct prefix. This allows user programs to describe custom structs out to tools, if required.

For example, a struct in C that looks like this:

```
struct mytype {
  char data[20];
```

```
};
```

Would be represented by the following field:

```
struct mytype myname 20
```

Deleting

Deleting an event from within a user process is done via ioctl() out to the /sys/kernel/debug/tracing/user_events_data file. The command to issue is DIAG_IOCSDEL.

This command only requires a single string specifying the event to delete by its name. Delete will only succeed if there are no references left to the event (in both user and kernel space). User programs should use a separate file to request deletes than the one used for registration due to this.

Status

When tools attach/record user based events the status of the event is updated in realtime. This allows user programs to only incur the cost of the write() or writev() calls when something is actively attached to the event.

User programs call mmap() on /sys/kernel/debug/tracing/user_events_status to check the status for each event that is registered. The byte to check in the file is given back after the register ioctl() via user_reg.status_index. Currently the size of user_events_status is a single page, however, custom kernel configurations can change this size to allow more user based events. In all cases the size of the file is a multiple of a page size.

For example, if the register ioctl() gives back a status_index of 3 you would check byte 3 of the returned mmap data to see if anything is attached to that event.

Administrators can easily check the status of all registered events by reading the user_events_status file directly via a terminal. The output is as follows:

```
Byte:Name [# Comments]
...

Active: ActiveCount
Busy: BusyCount
Max: MaxCount
```

For example, on a system that has a single event the output looks like this:

```
1:test

Active: 1
Busy: 0
Max: 4096
```

If a user enables the user event via ftrace, the output would change to this:

```
1:test # Used by ftrace

Active: 1
Busy: 1
Max: 4096
```

NOTE: *A status index of 0 will never be returned. This allows user programs to have an index that can be used on error cases.*

Status Bits

The byte being checked will be non-zero if anything is attached. Programs can check specific bits in the byte to see what mechanism has been attached.

The following values are defined to aid in checking what has been attached:

EVENT_STATUS_FTRACE - Bit set if ftrace has been attached (Bit 0).

EVENT_STATUS_PERF - Bit set if perf has been attached (Bit 1).

Writing Data

After registering an event the same fd that was used to register can be used to write an entry for that event. The write_index returned must be at the start of the data, then the remaining data is treated as the payload of the event.

For example, if write_index returned was 1 and I wanted to write out an int payload of the event. Then the data would have to be 8 bytes (2 ints) in size, with the first 4 bytes being equal to 1 and the last 4 bytes being equal to the value I want as the payload.

In memory this would look like this:

```
int index;
int payload;
```

User programs might have well known structs that they wish to use to emit out as payloads. In those cases writev() can be used, with the first vector being the index and the following vector(s) being the actual event payload.

For example, if I have a struct like this:

```
struct payload {
      int src;
      int dst;
      int flags;
};
```

It's advised for user programs to do the following:

```
struct iovec io[2];
struct payload e;

io[0].iov_base = &write_index;
io[0].iov_len = sizeof(write_index);
io[1].iov_base = &e;
io[1].iov_len = sizeof(e);

writev(fd, (const struct iovec*)io, 2);
```

NOTE: *The write_index is not emitted out into the trace being recorded.*

Example Code

See sample code in samples/user_events.