libbpf

unknown

Dec 21, 2022

CONTENTS

CHAPTER

ONE

PROGRAM TYPES AND ELF SECTIONS

The table below lists the program types, their attach types where relevant and the ELF section names supported by libbpf for them. The ELF section names follow these rules:

- `type` is an exact match, e.g. `SEC("socket")`
- `type+` means it can be either exact `SEC("type")` or well-formed `SEC("type/extras")` with a '/' separator between `type` and `extras`.

When `extras` are specified, they provide details of how to auto-attach the BPF program. The format of `extras` depends on the program type, e.g. `SEC("tracepoint/<category>/<name>")` for tracepoints or `SEC("usdt/<path>:<provider>:<name>")` for USDT probes. The extras are described in more detail in the footnotes.

Program Type	Attach Type	ELF Section Name	Sleepable
BPF_PROG_TYPE_CGROUP_DEVICE	BPF_CGROUP_DEVICE	cgroup/dev	
BPF_PROG_TYPE_CGROUP_SKB		cgroup/skb	
	BPF_CGROUP_INET_EGRESS	cgroup_skb/egress	
	BPF_CGROUP_INET_INGRESS	cgroup_skb/ingress	
BPF_PROG_TYPE_CGROUP_SOCKOPT	BPF_CGROUP_GETSOCKOPT	cgroup/getsockopt	
	BPF_CGROUP_SETSOCKOPT	cgroup/setsockopt	
BPF_PROG_TYPE_CGROUP_SOCK_ADDR	BPF_CGROUP_INET4_BIND	cgroup/bind4	
	BPF_CGROUP_INET4_CONNECT	cgroup/connect4	
	BPF_CGROUP_INET4_GETPEERNAME	cgroup/getpeername4	
	BPF_CGROUP_INET4_GETSOCKNAME	cgroup/getsockname4	
	BPF_CGROUP_INET6_BIND	cgroup/bind6	
	BPF_CGROUP_INET6_CONNECT	cgroup/connect6	
	BPF_CGROUP_INET6_GETPEERNAME	cgroup/getpeername6	
	BPF_CGROUP_INET6_GETSOCKNAME	cgroup/getsockname6	
	BPF_CGROUP_UDP4_RECVMSG	cgroup/recvmsg4	
	BPF_CGROUP_UDP4_SENDMSG	cgroup/sendmsg4	
	BPF_CGROUP_UDP6_RECVMSG	cgroup/recvmsg6	
	BPF_CGROUP_UDP6_SENDMSG	cgroup/sendmsg6	
BPF_PROG_TYPE_CGROUP_SOCK	BPF_CGROUP_INET4_POST_BIND	cgroup/post_bind4	
	BPF_CGROUP_INET6_POST_BIND	cgroup/post_bind6	
	BPF_CGROUP_INET_SOCK_CREATE	cgroup/sock_create	
		cgroup/sock	
	BPF_CGROUP_INET_SOCK_RELEASE	cgroup/sock_release	
BPF_PROG_TYPE_CGROUP_SYSCTL	BPF_CGROUP_SYSCTL	cgroup/sysctl	
BPF_PROG_TYPE_EXT		freplace+[1]	
BPF_PROG_TYPE_FLOW_DISSECTOR	BPF_FLOW_DISSECTOR	flow_dissector	
BPF_PROG_TYPE_KPROBE		kprobe+[2]	
		kretprobe+[Page 3, 2]	

continues on next page

Table 1 – continued from previous page

Program Type	Attach Type	ELF Section Name	Sleepable
		ksyscall+[3]	
		kretsyscall+[Page 3, 3]	
		uprobe+[4]	
		uprobe.s+[Page 3, 4]	Yes
		uretprobe+[Page 3, 4]	
		uretprobe.s+[Page 3, 4]	Yes
		usdt+[5]	
	BPF_TRACE_KPROBE_MULTI	kprobe.multi+[6]	
		kretprobe.multi+[Page 3, 6]	
BPF_PROG_TYPE_LIRC_MODE2	BPF_LIRC_MODE2	lirc_mode2	
BPF_PROG_TYPE_LSM	BPF_LSM_CGROUP	lsm_cgroup+	
	BPF_LSM_MAC	lsm+[7]	
		lsm.s+[Page 3, 7]	Yes
BPF_PROG_TYPE_LWT_IN		lwt_in	
BPF_PROG_TYPE_LWT_OUT		lwt_out	
BPF_PROG_TYPE_LWT_SEG6LOCAL		lwt_seg6local	
BPF_PROG_TYPE_LWT_XMIT		lwt_xmit	
BPF_PROG_TYPE_PERF_EVENT		perf_event	
BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE		raw_tp.w+[8]	
		raw_tracepoint.w+	
BPF_PROG_TYPE_RAW_TRACEPOINT		raw_tp+[Page 3, 8]	
		raw_tracepoint+	
BPF_PROG_TYPE_SCHED_ACT		action	
BPF_PROG_TYPE_SCHED_CLS		classifier	
		tc	
BPF_PROG_TYPE_SK_LOOKUP	BPF_SK_LOOKUP	sk_lookup	
BPF_PROG_TYPE_SK_MSG	BPF_SK_MSG_VERDICT	sk_msg	
BPF_PROG_TYPE_SK_REUSEPORT	BPF_SK_REUSEPORT_SELECT_OR_MIGRATE	sk_reuseport/migrate	
	BPF_SK_REUSEPORT_SELECT	sk_reuseport	
BPF_PROG_TYPE_SK_SKB		sk_skb	
	BPF_SK_SKB_STREAM_PARSER	sk_skb/stream_parser	
	BPF_SK_SKB_STREAM_VERDICT	sk_skb/stream_verdict	
BPF_PROG_TYPE_SOCKET_FILTER		socket	
BPF_PROG_TYPE_SOCK_OPS	BPF_CGROUP_SOCK_OPS	sockops	
BPF_PROG_TYPE_STRUCT_OPS		struct_ops+	
BPF_PROG_TYPE_SYSCALL		syscall	Yes
BPF_PROG_TYPE_TRACEPOINT		tp+[9]	
		tracepoint+[Page 3, 9]	
BPF_PROG_TYPE_TRACING	BPF_MODIFY_RETURN	fmod_ret+[Page 3, 1]	
		fmod_ret.s+[Page 3, 1]	Yes
	BPF_TRACE_FENTRY	fentry+[Page 3, 1]	
		fentry.s+[Page 3, 1]	Yes
	BPF_TRACE_FEXIT	fexit+[Page 3, 1]	
		fexit.s+[Page 3, 1]	Yes
	BPF_TRACE_ITER	iter+[10]	
		iter.s+[Page 3, 10]	Yes

continues on next page

Table 1 – continued from previous page

Program Type	Attach Type	ELF Section Name	Sleepable
	BPF_TRACE_RAW_TP	tp_btf+[Page 3, 1]	
BPF_PROG_TYPE_XDP	BPF_XDP_CPUMAP	xdp.frags/cpumap	
		xdp/cpumap	
	BPF_XDP_DEVMAP	xdp.frags/devmap	
		xdp/devmap	
	BPF_XDP	xdp.frags	
		xdp	

[1] The fentry attach format is fentry[.s]/<function>.

[2] The kprobe attach format is kprobe/<function>[+<offset>]. Valid characters for function are a-zA-Z0-9_. and offset must be a valid non-negative integer.

[3] The ksyscall attach format is ksyscall/<syscall>.

[4] The uprobe attach format is uprobe[.s]/<path>:<function>[+<offset>].

[5] The usdt attach format is usdt/<path>:<provider>:<name>.

[6] The kprobe.multi attach format is kprobe.multi/<pattern> where pattern supports * and ? wildcards. Valid characters for pattern are a-zA-Z0-9_.*?.

[7] The lsm attachment format is lsm[.s]/<hook>.

[8] The raw_tp attach format is raw_tracepoint[.w]/<tracepoint>.

[9] The tracepoint attach format is tracepoint/<category>/<name>.

[10] The iter attach format is iter[.s]/<struct-name>.

CHAPTER

TWO

API NAMING CONVENTION

libbpf API provides access to a few logically separated groups of functions and types. Every group has its own naming convention described here. It's recommended to follow these conventions whenever a new function or type is added to keep libbpf API clean and consistent.

All types and functions provided by libbpf API should have one of the following prefixes: `bpf_`, `btf_`, `libbpf_`, `btf_dump_`, `ring_buffer_`, `perf_buffer_`.

2.1 System call wrappers

System call wrappers are simple wrappers for commands supported by sys_bpf system call. These wrappers should go to `bpf.h` header file and map one to one to corresponding commands.

For example `bpf_map_lookup_elem` wraps `BPF_MAP_LOOKUP_ELEM` command of sys_bpf, `bpf_prog_attach` wraps `BPF_PROG_ATTACH`, etc.

2.2 Objects

Another class of types and functions provided by libbpf API is "objects" and functions to work with them. Objects are high-level abstractions such as BPF program or BPF map. They're represented by corresponding structures such as `struct bpf_object`, `struct bpf_program`, `struct bpf_map`, etc.

Structures are forward declared and access to their fields should be provided via corresponding getters and setters rather than directly.

These objects are associated with corresponding parts of ELF object that contains compiled BPF programs.

For example `struct bpf_object` represents ELF object itself created from an ELF file or from a buffer, `struct bpf_program` represents a program in ELF object and `struct bpf_map` is a map.

Functions that work with an object have names built from object name, double underscore and part that describes function purpose.

For example `bpf_object__open` consists of the name of corresponding object, `bpf_object`, double underscore and `open` that defines the purpose of the function to open ELF file and create `bpf_object` from it.

All objects and corresponding functions other than BTF related should go to `libbpf.h`. BTF types and functions should go to `btf.h`.

2.3 Auxiliary functions

Auxiliary functions and types that don't fit well in any of categories described above should have `libbpf_` prefix, e.g. `libbpf_get_error` or `libbpf_prog_type_by_name`.

2.4 ABI

libbpf can be both linked statically or used as DSO. To avoid possible conflicts with other libraries an application is linked with, all non-static libbpf symbols should have one of the prefixes mentioned in API documentation above. See API naming convention to choose the right name for a new symbol.

2.5 Symbol visibility

libbpf follow the model when all global symbols have visibility "hidden" by default and to make a symbol visible it has to be explicitly attributed with `LIBBPF_API` macro. For example:

```
LIBBPF_API int bpf_prog_get_fd_by_id(__u32 id);
```

This prevents from accidentally exporting a symbol, that is not supposed to be a part of ABI what, in turn, improves both libbpf developer- and user-experiences.

2.6 ABI versionning

To make future ABI extensions possible libbpf ABI is versioned. Versioning is implemented by `libbpf.map` version script that is passed to linker.

Version name is `LIBBPF_` prefix + three-component numeric version, starting from `0.0.1`.

Every time ABI is being changed, e.g. because a new symbol is added or semantic of existing symbol is changed, ABI version should be bumped. This bump in ABI version is at most once per kernel development cycle.

For example, if current state of `libbpf.map` is:

```
LIBBPF_0.0.1 {
        global:
                bpf_func_a;
                bpf_func_b;
        local:
                \*;
};
```

, and a new symbol `bpf_func_c` is being introduced, then `libbpf.map` should be changed like this:

```
LIBBPF_0.0.1 {
        global:
                bpf_func_a;
                bpf_func_b;
        local:
                \*;
};
```

(continues on next page)

(continued from previous page)

```
LIBBPF_0.0.2 {
        global:
                bpf_func_c;
} LIBBPF_0.0.1;
```

, where new version `LIBBPF_0.0.2` depends on the previous `LIBBPF_0.0.1`.

Format of version script and ways to handle ABI changes, including incompatible ones, described in details in [1].

2.7 Stand-alone build

Under https://github.com/libbpf/libbpf there is a (semi-)automated mirror of the mainline's version of libbpf for a stand-alone build.

However, all changes to libbpf's code base must be upstreamed through the mainline kernel tree.

CHAPTER

THREE

API DOCUMENTATION CONVENTION

The libbpf API is documented via comments above definitions in header files. These comments can be rendered by doxygen and sphinx for well organized html output. This section describes the convention in which these comments should be formated.

Here is an example from btf.h:

```
/**
 * @brief **btf__new()** creates a new instance of a BTF object from the raw
 * bytes of an ELF's BTF section
 * @param data raw bytes
 * @param size number of bytes passed in `data`
 * @return new BTF object instance which has to be eventually freed with
 * **btf__free()**
 *
 * On error, error-code-encoded-as-pointer is returned, not a NULL. To extract
 * error code from such a pointer `libbpf_get_error()` should be used. If
 * `libbpf_set_strict_mode(LIBBPF_STRICT_CLEAN_PTRS)` is enabled, NULL is
 * returned on error instead. In both cases thread-local `errno` variable is
 * always set to error code as well.
 */
```

The comment must start with a block comment of the form '/**'.

The documentation always starts with a @brief directive. This line is a short description about this API. It starts with the name of the API, denoted in bold like so: **api_name**. Please include an open and close parenthesis if this is a function. Follow with the short description of the API. A longer form description can be added below the last directive, at the bottom of the comment.

Parameters are denoted with the @param directive, there should be one for each parameter. If this is a function with a non-void return, use the @return directive to document it.

3.1 License

libbpf is dual-licensed under LGPL 2.1 and BSD 2-Clause.

3.2 Links

[1] **https://www.akkadia.org/drepper/dsohowto.pdf**
(Chapter 3. Maintaining APIs and ABIs).

CHAPTER

FOUR

BUILDING LIBBPF

libelf and zlib are internal dependencies of libbpf and thus are required to link against and must be installed on the system for applications to work. pkg-config is used by default to find libelf, and the program called can be overridden with PKG_CONFIG.

If using pkg-config at build time is not desired, it can be disabled by setting NO_PKG_CONFIG=1 when calling make.

To build both static libbpf.a and shared libbpf.so:

```
$ cd src
$ make
```

To build only static libbpf.a library in directory build/ and install them together with libbpf headers in a staging directory root/:

```
$ cd src
$ mkdir build root
$ BUILD_STATIC_ONLY=y OBJDIR=build DESTDIR=root make install
```

To build both static libbpf.a and shared libbpf.so against a custom libelf dependency installed in /build/root/ and install them together with libbpf headers in a build directory /build/root/:

```
$ cd src
$ PKG_CONFIG_PATH=/build/root/lib64/pkgconfig DESTDIR=/build/root make
```

This is documentation for libbpf, a userspace library for loading and interacting with bpf programs.

All general BPF questions, including kernel functionality, libbpf APIs and their application, should be sent to bpf@vger.kernel.org mailing list. You can subscribe to the mailing list search its archive. Please search the archive before asking new questions. It very well might be that this was already addressed or answered before.