

Name

bpftool-gen – tool for BPF code-generation

SYNOPSIS

bpftool [*OPTIONS*] **gen** *COMMAND*

OPTIONS := { { **-j** | **--json** } [{ **-p** | **--pretty** }] | { **-d** | **--debug** } | { **-L** | **--use-loader** } | [{ **-S** | **--sign** } { **-k** <private_key.pem> } **-i** <certificate.x509> } }

COMMAND := { **object** | **skeleton** | **help** }

GEN COMMANDS

bpftool gen object *OUTPUT_FILE* *INPUT_FILE* [*INPUT_FILE*...]

bpftool gen skeleton *FILE* [**name** *OBJECT_NAME*]

bpftool gen subskeleton *FILE* [**name** *OBJECT_NAME*]

bpftool gen min_core_btf *INPUT* *OUTPUT* *OBJECT* [*OBJECT*...]

bpftool gen help

DESCRIPTION

bpftool gen object *OUTPUT_FILE* *INPUT_FILE* [*INPUT_FILE*...]

Statically link (combine) together one or more *INPUT_FILE*'s into a single resulting *OUTPUT_FILE*. All the files involved are BPF ELF object files.

The rules of BPF static linking are mostly the same as for user-space object files, but in addition to combining data and instruction sections, .BTF and .BTF.ext (if present in any of the input files) data are combined together. .BTF data is deduplicated, so all the common types across *INPUT_FILE*'s will only be represented once in the resulting BTF information.

BPF static linking allows to partition BPF source code into individually compiled files that are then linked into a single resulting BPF object file, which can be used to generate BPF skeleton (with **gen skeleton** command) or passed directly into **libbpf** (using **bpf_object__open()** family of APIs).

bpftool gen skeleton *FILE*

Generate BPF skeleton C header file for a given *FILE*.

BPF skeleton is an alternative interface to existing libbpf APIs for working with BPF objects. Skeleton code is intended to significantly shorten and simplify code to load and work with BPF programs from userspace side. Generated code is tailored to specific input BPF object *FILE*, reflecting its structure by listing out available maps, program, variables, etc. Skeleton eliminates the need to lookup mentioned components by name. Instead, if skeleton instantiation succeeds, they are populated in skeleton structure as valid libbpf types (e.g., **struct bpf_map** pointer) and can be passed to existing generic libbpf APIs.

In addition to simple and reliable access to maps and programs, skeleton provides a storage for BPF links (**struct bpf_link**) for each BPF program within BPF object. When requested, supported BPF programs will be automatically attached and resulting BPF links stored for further use by user in pre-allocated fields in skeleton struct. For BPF programs that can't be automatically attached by libbpf, user can attach them manually, but store resulting BPF link in per-program link field. All such set up links will be automatically destroyed on BPF skeleton destruction. This eliminates the need for users to manage links manually and rely on libbpf support to detach programs and free up resources.

Another facility provided by BPF skeleton is an interface to global variables of all supported kinds: mutable, read-only, as well as extern ones. This interface allows to pre-setup initial values of variables before BPF object is loaded and verified by kernel. For non-read-only variables, the

same interface can be used to fetch values of global variables on userspace side, even if they are modified by BPF code.

During skeleton generation, contents of source BPF object *FILE* is embedded within generated code and is thus not necessary to keep around. This ensures skeleton and BPF object file are matching 1-to-1 and always stay in sync. Generated code is dual-licensed under LGPL-2.1 and BSD-2-Clause licenses.

It is a design goal and guarantee that skeleton interfaces are interoperable with generic libbpf APIs. User should always be able to use skeleton API to create and load BPF object, and later use libbpf APIs to keep working with specific maps, programs, etc.

As part of skeleton, few custom functions are generated. Each of them is prefixed with object name. Object name can either be derived from object file name, i.e., if BPF object file name is **example.o**, BPF object name will be **example**. Object name can be also specified explicitly through name *OBJECT_NAME* parameter. The following custom functions are provided (assuming **example** as the object name):

- **example__open** and **example__open_opts**. These functions are used to instantiate skeleton. It corresponds to libbpf's **bpf_object__open()** API. **_opts** variants accepts extra **bpf_object__open_opts** options.
- **example__load**. This function creates maps, loads and verifies BPF programs, initializes global data maps. It corresponds to libbpf's **bpf_object__load()** API.
- **example__open_and_load** combines **example__open** and **example__load** invocations in one commonly used operation.
- **example__attach** and **example__detach**. This pair of functions allow to attach and detach, correspondingly, already loaded BPF object. Only BPF programs of types supported by libbpf for auto-attachment will be auto-attached and their corresponding BPF links instantiated. For other BPF programs, user can manually create a BPF link and assign it to corresponding fields in skeleton struct. **example__detach** will detach both links created automatically, as well as those populated by user manually.
- **example__destroy**. Detach and unload BPF programs, free up all the resources used by skeleton and BPF object.

If BPF object has global variables, corresponding structs with memory layout corresponding to global data section layout will be created. Currently supported ones are: *.data*, *.bss*, *.rodata*, and *.kconfig* structs/data sections. These data sections/structs can be used to set up initial values of variables, if set before **example__load**. Afterwards, if target kernel supports memory-mapped BPF arrays, same structs can be used to fetch and update (non-read-only) data from userspace, with same simplicity as for BPF side.

bpftool gen subskeleton *FILE*

Generate BPF subskeleton C header file for a given *FILE*.

Subskeletons are similar to skeletons, except they do not own the corresponding maps, programs, or global variables. They require that the object file used to generate them is already loaded into a *bpf_object* by some other means.

This functionality is useful when a library is included into a larger BPF program. A subskeleton for the library would have access to all objects and globals defined in it, without having to know about the larger program.

Consequently, there are only two functions defined for subskeletons:

- **example__open(bpf_object*)**. Instantiates a subskeleton from an already opened (but not necessarily loaded) **bpf_object**.
- **example__destroy()**. Frees the storage for the subskeleton but *does not* unload any BPF programs or maps.

bpftool gen min_core_btf INPUT OUTPUT OBJECT [OBJECT...]

Generate a minimum BTF file as *OUTPUT*, derived from a given *INPUT* BTF file, containing all needed BTF types so one, or more, given eBPF objects CO-RE relocations may be satisfied.

When kernels aren't compiled with CONFIG_DEBUG_INFO_BTF, libbpf, when loading an eBPF object, has to rely on external BTF files to be able to calculate CO-RE relocations.

Usually, an external BTF file is built from existing kernel DWARF data using pahole. It contains all the types used by its respective kernel image and, because of that, is big.

The min_core_btf feature builds smaller BTF files, customized to one or multiple eBPF objects, so they can be distributed together with an eBPF CO-RE based application, turning the application portable to different kernel versions.

Check examples below for more information on how to use it.

bpftool gen help

Print short help message.

OPTIONS

-h, --help

Print short help message (similar to **bpftool help**).

-V, --version

Print bpftool's version number (similar to **bpftool version**), the number of the libbpf version in use, and optional features that were included when bpftool was compiled. Optional features include linking against LLVM or libbfd to provide the disassembler for JIT-ted programs (**bpftool prog dump jited**) and usage of BPF skeletons (some features like **bpftool prog profile** or showing pids associated to BPF objects may rely on it).

-j, --json

Generate JSON output. For commands that cannot produce JSON, this option has no effect.

-p, --pretty

Generate human-readable JSON output. Implies **-j**.

-d, --debug

Print all logs available, even debug-level information. This includes logs from libbpf as well as from the verifier, when attempting to load programs.

-L, --use-loader

For skeletons, generate a "light" skeleton (also known as "loader" skeleton). A light skeleton contains a loader eBPF program. It does not use the majority of the libbpf infrastructure, and does not need libelf.

-S, --sign

For skeletons, generate a signed skeleton. This option must be used with **-k** and **-i**. Using this flag implicitly enables **--use-loader**.

-k <private_key.pem>

Path to the private key file in PEM format, required for signing.

-i <certificate.x509>

Path to the X.509 certificate file in PEM or DER format, required for signing.

EXAMPLES**\$ cat example1.bpf.c**

```
#include <stdbool.h>
#include <linux/ptrace.h>
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

const volatile int param1 = 42;
bool global_flag = true;
struct { int x; } data = {};

SEC("raw_tp/sys_enter")
int handle_sys_enter(struct pt_regs *ctx)
{
    static long my_static_var;
    if (global_flag)
        my_static_var++;
    else
        data.x += param1;
    return 0;
}
```

\$ cat example2.bpf.c

```
#include <linux/ptrace.h>
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 128);
    __type(key, int);
    __type(value, long);
} my_map SEC(".maps");

SEC("raw_tp/sys_exit")
int handle_sys_exit(struct pt_regs *ctx)
{
    int zero = 0;
    bpf_map_lookup_elem(&my_map, &zero);
    return 0;
}
```

\$ cat example3.bpf.c

```
#include <linux/ptrace.h>
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
/* This header file is provided by the bpf_testmod module. */
#include "bpf_testmod.h"

int test_2_result = 0;
```

```

/* bpf_Testmod.ko calls this function, passing a "4"
 * and testmod_map->data.
 */
SEC("struct_ops/test_2")
void BPF_PROG(test_2, int a, int b)
{
    test_2_result = a + b;
}

SEC(".struct_ops")
struct bpf_testmod_ops testmod_map = {
    .test_2 = (void *)test_2,
    .data = 0x1,
};

```

This is example BPF application with three BPF programs and a mix of BPF maps and global variables. Source code is split across three source code files.

```
$ clang --target=bpf -g example1.bpf.c -o example1.bpf.o
```

```
$ clang --target=bpf -g example2.bpf.c -o example2.bpf.o
```

```
$ clang --target=bpf -g example3.bpf.c -o example3.bpf.o
```

```
$ bpftool gen object example.bpf.o example1.bpf.o example2.bpf.o example3.bpf.o
```

This set of commands compiles *example1.bpf.c*, *example2.bpf.c* and *example3.bpf.c* individually and then statically links respective object files into the final BPF ELF object file *example.bpf.o*.

```
$ bpftool gen skeleton example.bpf.o name example | tee example.skel.h
```

```

/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */

/* THIS FILE IS AUTOGENERATED! */
#ifndef __EXAMPLE_SKEL_H__
#define __EXAMPLE_SKEL_H__

#include <stdlib.h>
#include <bpf/libbpf.h>

struct example {
    struct bpf_object_skeleton *skeleton;
    struct bpf_object *obj;
    struct {
        struct bpf_map *rodata;
        struct bpf_map *data;
        struct bpf_map *bss;
        struct bpf_map *my_map;
        struct bpf_map *testmod_map;
    } maps;
    struct {
        struct example__testmod_map__bpf_testmod_ops {
            const struct bpf_program *test_1;
            const struct bpf_program *test_2;

```

```

        int data;
    } *testmod_map;
} struct_ops;
struct {
    struct bpf_program *handle_sys_enter;
    struct bpf_program *handle_sys_exit;
} progs;
struct {
    struct bpf_link *handle_sys_enter;
    struct bpf_link *handle_sys_exit;
} links;
struct example__bss {
    struct {
        int x;
    } data;
    int test_2_result;
} *bss;
struct example__data {
    _Bool global_flag;
    long int handle_sys_enter_my_static_var;
} *data;
struct example__rodata {
    int param1;
} *rodata;
};

static void example__destroy(struct example *obj);
static inline struct example *example__open_opts(
    const struct bpf_object_open_opts *opts);
static inline struct example *example__open();
static inline int example__load(struct example *obj);
static inline struct example *example__open_and_load();
static inline int example__attach(struct example *obj);
static inline void example__detach(struct example *obj);

#endif /* __EXAMPLE_SKEL_H__ */

```

\$ cat example.c

```

#include "example.skel.h"

int main()
{
    struct example *skel;
    int err = 0;

    skel = example__open();
    if (!skel)
        goto cleanup;

    skel->rodata->param1 = 128;

    /* Change the value through the pointer of shadow type */
    skel->struct_ops.testmod_map->data = 13;
}

```

```

err = example__load(skel);
if (err)
    goto cleanup;

/* The result of the function test_2() */
printf("test_2_result: %d\n", skel->bss->test_2_result);

err = example__attach(skel);
if (err)
    goto cleanup;

/* all libbpf APIs are usable */
printf("my_map name: %s\n", bpf_map__name(skel->maps.my_map));
printf("sys_enter prog FD: %d\n",
        bpf_program__fd(skel->progs.handle_sys_enter));

/* detach and re-attach sys_exit program */
bpf_link__destroy(skel->links.handle_sys_exit);
skel->links.handle_sys_exit =
    bpf_program__attach(skel->progs.handle_sys_exit);

printf("my_static_var: %ld\n",
        skel->bss->handle_sys_enter_my_static_var);

cleanup:
    example__destroy(skel);
    return err;
}

```

./example

```

test_2_result: 17
my_map name: my_map
sys_enter prog FD: 8
my_static_var: 7

```

This is a stripped-out version of skeleton generated for above example code.

min_core_btf

\$ bpftool btf dump file 5.4.0-example.btf format raw

```

[1] INT 'long unsigned int' size=8 bits_offset=0 nr_bits=64 encoding=(none)
[2] CONST '(anon)' type_id=1
[3] VOLATILE '(anon)' type_id=1
[4] ARRAY '(anon)' type_id=1 index_type_id=21 nr_elems=2
[5] PTR '(anon)' type_id=8
[6] CONST '(anon)' type_id=5
[7] INT 'char' size=1 bits_offset=0 nr_bits=8 encoding=(none)
[8] CONST '(anon)' type_id=7
[9] INT 'unsigned int' size=4 bits_offset=0 nr_bits=32 encoding=(none)
<long output>

```

\$ bpftool btf dump file one.bpf.o format raw

```

[1] PTR '(anon)' type_id=2

```

```

[2] STRUCT 'trace_event_raw_sys_enter' size=64 vlen=4
    'ent' type_id=3 bits_offset=0
    'id' type_id=7 bits_offset=64
    'args' type_id=9 bits_offset=128
    '__data' type_id=12 bits_offset=512
[3] STRUCT 'trace_entry' size=8 vlen=4
    'type' type_id=4 bits_offset=0
    'flags' type_id=5 bits_offset=16
    'preempt_count' type_id=5 bits_offset=24
<long output>

```

\$ bpftool gen min_core_btf 5.4.0-example.btf 5.4.0-smaller.btf one.bpf.o

\$ bpftool btf dump file 5.4.0-smaller.btf format raw

```

[1] TYPEDEF 'pid_t' type_id=6
[2] STRUCT 'trace_event_raw_sys_enter' size=64 vlen=1
    'args' type_id=4 bits_offset=128
[3] STRUCT 'task_struct' size=9216 vlen=2
    'pid' type_id=1 bits_offset=17920
    'real_parent' type_id=7 bits_offset=18048
[4] ARRAY '(anon)' type_id=5 index_type_id=8 nr_elems=6
[5] INT 'long unsigned int' size=8 bits_offset=0 nr_bits=64 encoding=(none)
[6] TYPEDEF '__kernel_pid_t' type_id=8
[7] PTR '(anon)' type_id=3
[8] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
<end>

```

Now, the "5.4.0-smaller.btf" file may be used by libbpf as an external BTF file when loading the "one.bpf.o" object into the "5.4.0-example" kernel. Note that the generated BTF file won't allow other eBPF objects to be loaded, just the ones given to min_core_btf.

```

LIBBPF_OPTS(bpf_object_open_opts, opts, .btf_custom_path = "5.4.0-smaller.btf");
struct bpf_object *obj;

obj = bpf_object__open_file("one.bpf.o", &opts);

...

```

SEE ALSO

bpf(2), bpf-helpers(7), bpftool(8), bpftool-btf(8), bpftool-cgroup(8), bpftool-feature(8), bpftool-iter(8), bpftool-link(8), bpftool-map(8), bpftool-net(8), bpftool-perf(8), bpftool-prog(8), bpftool-struct_ops(8), bpftool-token(8)