Workgroup: Network Working Group
Internet-Draft: draft-thaler-bpf-elf-00
Published: 13 March 2023
Intended Status: Standards Track
Expires: 14 September 2023
Authors: D. Thaler, Ed.
Microsoft

eBPF ELF Profile Specification, v0.1

Abstract

The Executable and Linking Format (ELF) is specified in Chapter 4 of the System V Application Binary Interface. This document specifies version 0.1 of the eBPF profile for ELF files.

Status of This Memo

This Internet-Draft will expire on 14 September 2023.

Copyright Notice

Table of Contents

1. Documentation conventions

This specification is a extension to the ELF file format as specified in Chapter 4 of the System V Application Binary Interface [ELF]. As such, the same data representation convention is used as specified in the Data Representation section of the ELF specification, where structures are represented in a C-style format with types such as Elf64_Word for an unsigned 64-bit integer.

NOTE: Some content in this draft will eventually move to a separate BTF draft.

2. ELF Header

The ELF header must have values set as follows:

*e_ident[EI_CLASS] must be set to ELFCLASS64 (2).

*e_type must be set to ET_REL (1).

*e_machine must be set to EM_BPF (247).

3. TEXT Sections

eBPF programs [BPF-ISA] are stored in TEXT sections. A TEXT section can contain multiple eBPF programs, each with a different program name which is stored as a function in a TEXT section. The ".text" section can be empty if eBPF programs are stored in other TEXT sections.

This specification does not mandate any particular convention for TEXT section names, as there are multiple different conventions in use today, including:

- Prefix Convention: The section name is prefixed with a string that identifies the program type, so that the program type of any programs in the section can be determined by finding the longest substring match across all program type prefixes.

- Exact Match Convention: The section name is a string that identifies the program type of any programs in the section.

- Arbitrary Convention: The section name can be anything and the program type of any programs in the section must be determined without consulting the section name.

4. DATA Sections

4.1. Classic Map Definitions

Classic eBPF map definitions are stored in DATA sections named "maps" or matching "maps/<map-name>". Each such section can contain 0 or more map definitions. The number of map definitions in a section can be determined by counting the number of symbols in the ".symtab" section that point into that maps section.

The size of a map definition can be calculated as:

(size of maps section) / (count of map definitions in that section)

The format of a map definition is as follows, where fields are in the byte order indicated in e_ident[EI_DATA] in the ELF header:

```
 typedef struct {
    Elf64_Word    type;
    Elf64_Word    key_size;
    Elf64_Word    value_size;
    Elf64_Word    max_entries;
    Elf64_Word    inner_map_idx;
    unsigned char platform_specific_data[];
 } Elf64_BpfMapDefinition;
```

type An integer identifying the map type. Its value and meaning are platform-specific.

key_size Size in bytes of keys in the map, if any.

value_size
Size in bytes of values in the map, if any.

max_entries Maximum number of entries in the map, if the map type has a maximum.

inner_map_idx If the map type is one whose values contain ids of other maps, then the inner map index must be set to the 0-based index of another map definition in the section. The referenced map definition is used to enforce that any maps must match it for their ids to be allowed as values of this map. If the map type is not one whose values contain ids of other maps, this must be set to 0.

platform_specific_data This field and its size is up to the runtime platform to define. For example, on Linux 4.14 and later, this can hold a NUMA node value.

4.2. BTF Map Definitions

BTF eBPF map definitions are stored in a DATA section named ".maps". The number of map definitions in a section can be determined by counting the number of symbols in the ".symtab" section that point into the ".maps" section.

TODO: add format description here

5. Other Sections

section name	reference
license	Program License (Section 5.1)
version	Runtime Version restriction (Section 5.2)
.BTF	Type and String Data (Section 5.3)
.BTF.ext	Function and Line Information (Section 5.4)
.BTF_ids	BTF ID Values (Section 5.5)

Table 1

5.1. Program License

A runtime can optionally restrict what program types and/or helper functions can be used based on what license the eBPF program is under. This information can be placed into the ELF file in a section named "license" whose contents is a null-terminated SPDX license expression as specified in Annex D of ISO/IEC 5962:2021, "Information technology -- SPDX(R) Specification V22.1 [SPDX].

5.2. Runtime Version restriction

A runtime can optionally restrict whether an eBPF program can load based on what runtime version it was designed to interact with. This information can be placed into the ELF file in a section named "version" containing a 4-byte version identifier whose use is runtime-specific.

5.3. Type and String Data

The optional ".BTF" section contains type and string data. The format of this section is the same as specified in BTF Type and String Encoding [BTF].

5.4. Function and Line Information

The optional ".BTF.ext" section contains source line information for the first eBPF instruction for each source line.

The section starts with the following header:

```
typedef struct {
   Elf64_Half    magic;
   unsigned char version;
   unsigned char flags;
   Elf64_Word    hdr_len;
   Elf64_Word    func_info_off;
   Elf64_Word    func_info_len;
   Elf64_Word    line_info_off;
   Elf64_Word    line_info_len;
   unsigned char platform_specific_data[];
} Elf64_BtfExtHeader;
```

magic
Must be set to 0xeB9F, which can be used by a parser to determine whether multi-byte fields are in little-endian or big-endian byte order.

version Must be set to 1 (0x01).

flags Must be set to 0.

hdr_len The size in bytes of this structure including the platform_specific_data.

func_info_off Offset in bytes past the end of the header, of the start of the Function information (Section 5.4.1).

func_info_len Size in bytes of the Function information (Section 5.4.1). Must be set to 8 (0x00000008).

line_info_off Offset in bytes past the end of the header, of the start of the Line Information (Section 5.4.4).

line_info_len Size in bytes of the Line Information (Section 5.4.4). Must be set to 16 (0x00000010).

platform_specific_data This field and its size is up to the runtime platform to define.

5.4.1. Function information

```
typedef struct {
    Elf64_Word          func_info_rec_size;
    Elf64_BtfExtInfoSec btf_ext_info_sec[];
} Elf64_BpfFunctionInfo;
```

func_info_rec_size Size in bytes of each function record contained in an Info block (Section 5.4.2). Must be set to 8 (0x00000008).

btf_ext_info_sec A set of Info block (Section 5.4.2) data blobs, as many as will fit in the size given as the func_info_len, where each record within an info block is formatted as shown under Function Record (Section 5.4.3) below.

5.4.2. Info block

```
typedef struct {
   Elf64_Word    sec_name_off;
   Elf64_Word    num_info;
   unsigned char data[];
} Elf64_BtfExtInfoSec;
```

sec_name_off Offset in bytes of the section name within the Type and String Data (Section 5.3).

num_info Number of records that follow. Must be greater than 0.

data A series of function or line records. The total length of data is num_info * record_size bytes, where record_size is the size of a function record or line record.

5.4.3. Function Record

```
typedef struct {
    Elf64_Word insn_off;
    Elf64_Word type_id;
} Elf64_BpfFunctionInfo;
```

insn_off Number 8 byte units from the start of the section whose name is given by "Section name offset" to the start of the function. Must be 0 for the first record, and for subsequent records it must be greater than the instruction offset of the previous record.

type_id TODO: Add a definition of this field, which is "a BTF_KIND_FUNC type".

5.4.4. Line Information

```
typedef struct {
    Elf64_Word          line_info_rec_size;
    Elf64_BtfExtInfoSec  btf_ext_info_sec[];
} Elf64_BpfLineInfo;
```

line_info_rec_size Size in bytes of each line record in an Info block (Section 5.4.2). Must be set to 16 (0x00000010).

btf_ext_info_sec A set of Info block (Section 5.4.2) data blobs, as many as will fit in the size given as the line_info_len, where each record within an info block is formatted as shown under Line Record (Section 5.4.5) below.

5.4.5. Line Record

```
typedef struct {
    Elf64_Word insn_off;
    Elf64_Word file_name_off;
    Elf64_Word line_off;
    Elf64_Word line_col;
} ELF32_BpfLineInfo;
```

insn_off 0-based instruction index into the eBPF program contained in the section whose name is referenced in the Info block (Section 5.4.2).

file_name_off Offset in bytes of the file name within the Type and String Data (Section 5.3).

line_off Offset in bytes of the source line within the Type and String Data (Section 5.3).

line_col The line and column number value, computed as (line number << 10) | (column number).

5.5. BTF ID Values

TODO: make this secction adhere to the ELF specification data format

The .BTF_ids section encodes BTF ID values that are used within the Linux kernel.

This section is created during the Linux kernel compilation with the help of macros defined in include/linux/btf_ids.h header file. Kernel code can use them to create lists and sets (sorted lists) of BTF ID values.

The BTF_ID_LIST and BTF_ID macros define unsorted list of BTF ID values, with following syntax:

```
BTF_ID_LIST(list)
BTF_ID(type1, name1)
BTF_ID(type2, name2)
```

resulting in the following layout in the .BTF_ids section:

```
__BTF_ID__type1__name1__1:
.zero 4
__BTF_ID__type2__name2__2:
.zero 4
```

The u32 list[] variable is defined to access the list.

The BTF_ID_UNUSED macro defines 4 zero bytes. It's used when we want to define an unused entry in BTF_ID_LIST, like:

```
BTF_ID_LIST(bpf_skb_output_btf_ids)
BTF_ID(struct, sk_buff)
BTF_ID_UNUSED
BTF_ID(struct, task_struct)
```

The BTF_SET_START/END macros pair defines a sorted list of BTF ID values and their count, with following syntax:

```
BTF_SET_START(set)
BTF_ID(type1, name1)
BTF_ID(type2, name2)
BTF_SET_END(set)
```

resulting in the following layout in the .BTF_ids section:

```
__BTF_ID__set__set:
.zero 4
__BTF_ID__type1__name1__3:
.zero 4
__BTF_ID__type2__name2__4:
.zero 4
```

The struct btf_id_set set; variable is defined to access the list.

The typeX name can be one of following:

struct, union, typedef, func

and is used as a filter when resolving the BTF ID value.

All the BTF ID lists and sets are compiled in the .BTF_ids section and resolved during the linking phase of Linux kernel build by resolve_btfids tool.

6. Acknowledgements

Portions of this draft were derived from information in btf.rst in the Linux kernel repository, to which a number of individuals have contributed over time, including Andrii Nakryiko, Dave Tucker, David S. Miller, Gary Lin, Ilya Leoshkevich, Indu Bhagat, Jesper Dangaard Brouer, Jiri Olsa, Jonathan Corbet, Mauro Carvalho Chehab, Rong Tao, and Yonghong Song.

7. Normative References

[BPF-ISA] Thaler, D., Ed., "eBPF Instruction Set Specification, v1.0", Work in Progress, Internet-Draft, draft-thaler-bpf-isa-00, 13 March 2023, <https://datatracker.ietf.org/doc/html/draft-thaler-bpf-isa-00>.

[BTF] "BPF Type Format (BTF)", <https://elixir.bootlin.com/linux/latest/source/Documentation/bpf/btf.rst>.

[ELF] "System V Application Binary Interface", <http://www.sco.com/developers/gabi/latest/contents.html>.

[SPDX]
"Information technology -- SPDX® Specification V.2.1", <https://www.iso.org/standard/81870.html>.

Author's Address

Dave Thaler (editor)
Microsoft
Redmond, WA 98052
United States of America

Email: dthaler@microsoft.com