Rootless な環境における eBPF の活用

松本直樹(@PiBVT)

2025/05/30

eBPF Japan Meetup#4

スライド: https://pibvt.net/s/zUEhT3GJrUc

背景: eBPF をもっと便利に活用

本来の “Packet Filter” 以外にトレーシング等でも活用されるように

安全な活用・運用という面は？

- 「eBPF はカーネル内で安全にプログラムが動かせる」
- 「Verifier が頑張るから安全」

対応するユーザープロセスはあまり気にされていない

- 「とりえあず sudo, root で実行」
- 「eBPF だから `--privileged` が必要」

→ユーザープロセスに脆弱性があった場合被害拡大の恐れ

最小権限の原則(PoLP)は eBPF においても例外ではない

どんな権限が eBPF の利用には必要？

スライド: https://pibvt.net/s/zUEhT3GJrUc

本発表の流れ

eBPF の利用に必要な権限とその関係は実は複雑

「**非特権プロセスや(Rootless)コンテナで eBPF は使えるのか？**」

目次

1. eBPF が動作するまでの流れと要求する権限
   各種 Program Type ごとに(Socket, XDP, Kprobe, Tracepoint, Uprobe)
2. BPF Token による権限管理
3. (Rootless)コンテナにおける eBPF の活用例
4. まとめ

スライド: https://pibvt.net/s/zUEhT3GJrUc

本発表の流れ

eBPF の利用に必要な権限とその関係は実は複雑

「**非特権プロセスや(Rootless)コンテナで eBPF は使えるのか？**」

目次

8割
1. eBPF が動作するまでの流れと要求する権限
   各種 Program Type ごとに(Socket, XDP, Kprobe, Tracepoint, Uprobe)

1割
2. BPF Token による権限管理

1割
3. (Rootless)コンテナにおける eBPF の活用例

4. まとめ

スライド: https://pibvt.net/s/zUEhT3GJrUc

想定環境/参考文献

本発表では以下の環境を想定

- OS: Ubuntu 25.04
- Linux Kernel: 6.14.6

参考文献

- eBPFのRootlessコンテナでの応用の可能性について調べてみた https://zenn.dev/yutarohayakawa/articles/d8dc9992d9604e
- eBPF 導入のためのセキュリティ検討 https://speakerdeck.com/kentatada/security-for-introducing-ebpf
- eBPF Security Threat Model https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20—%20eBPF%20Security%20Threat%20Model.pdf

スライド: https://pibvt.net/s/zUEhT3GJrUc

本発表の流れ

eBPF の利用に必要な権限とその関係は実は複雑

「**非特権プロセスや(Rootless)コンテナで eBPF は使えるのか？**」

目次

1. eBPF が動作するまでの流れと要求する権限
   - 各種 Program Type ごとに(Socket, XDP, Kprobe Tracepoint, Uprobe)
2. BPF Token による権限管理
3. (Rootless)コンテナにおける eBPF の活用例
4. まとめ

eBPF の活用領域

eBPF の活用領域は多岐にわたる

- **ネットワーク系**
  - Socket: Raw socket でのフィルタリング等
  - XDP, SKB, TC でのパケットのフィルタリングや改変
  - Lightweight Tunnel: トンネリングに関するフレームワーク
- **トレーシング系**
  - Kprobe, Tracepoint: カーネル内部の挙動をトレース
  - Uprobe: ユーザープロセスの挙動をトレース
- cgroup: cgroup 単位で SKB や Socket の挙動を制御

参考: https://docs.ebpf.io/linux/program-type/
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF の活用領域

eBPF の活用領域は多岐にわたる

- **ネットワーク系**
  - Socket: Raw socket でのフィルタリング等
  - XDP, SKB, TC でのパケットのフィルタリングや改変
  - Lightweight Tunnel: トンネリングに関するフレームワーク
- **トレーシング系**
  - Kprobe, Tracepoint: カーネル内部の挙動をトレース
  - Uprobe: ユーザープロセスの挙動をトレース
- cgroup: cgroup 単位で SKB や Socket の挙動を制御

参考: https://docs.ebpf.io/linux/program-type/
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF 動作の流れ(Socket)

Socket に対して eBPF プログラムをアタッチする linux/samples/bpf/sockex1 を利用

- eBPF プログラム: パケット長をカウントし、MAP に格納
- User: RAW SOCKET に attach し、bpf_map からカウントを読み出し

```
struct {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, u32);
        __type(value, long);
        __uint(max_entries, 256);
} my_map SEC(".maps");

SEC("socket1")
int bpf_prog1(struct __sk_buff *skb)
{
        int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
        long *value;

        if (skb->pkt_type != PACKET_OUTGOING)
                return 0;

        value = bpf_map_lookup_elem(&my_map, &index);
        if (value)
                __sync_fetch_and_add(value, skb->len);

        return 0;
}
char _license[] SEC("license") = "GPL";
```

https://elixir.bootlin.com/linux/v6.14.6/source/samples/bpf/sockex1_kern.c

```
        snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);

        obj = bpf_object__open_file(filename, NULL);
        if (libbpf_get_error(obj))
                return 1;

        prog = bpf_object__next_program(obj, NULL);
        bpf_program__set_type(prog, BPF_PROG_TYPE_SOCKET_FILTER);

        err = bpf_object__load(obj);
        if (err)
                return 1;

        prog_fd = bpf_program__fd(prog);
        map_fd = bpf_object__find_map_fd_by_name(obj, "my_map");

        sock = open_raw_sock("lo");

        assert(setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd,
                          sizeof(prog_fd)) == 0);
```

https://elixir.bootlin.com/linux/v6.14.6/source/samples/bpf/sockex1_user.c

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(Socket)

User のプログラムは動作に特権を要求

「eBPF を使うためには高い権限が必要、とりあえず特権で」

→ 不必要に高い権限を付与 = リスクのある使い方

どこで**どういった権限**が要求されるのか？

```
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./sockex1
libbpf: Failed to bump RLIMIT_MEMLOCK (err = -EPERM), you might need to do i
t explicitly!
libbpf: Error in bpf_object__probe_loading(): -EPERM. Couldn't load trivial
BPF program. Make sure your kernel supports BPF (CONFIG_BPF_SYSCALL=y) and/o
r that RLIMIT_MEMLOCK is set to big enough value.
libbpf: failed to load object './sockex1_kern.o'
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo ./sockex1
TCP 0 UDP 0 ICMP 0 bytes
TCP 184 UDP 0 ICMP 196 bytes
TCP 184 UDP 0 ICMP 392 bytes
TCP 585 UDP 0 ICMP 588 bytes
TCP 585 UDP 0 ICMP 784 bytes
```

← 一般ユーザーでは動かない

← 特権ユーザーなら動く

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(Socket)

BPF_PROG_LOAD が **EPERM(Operation not permitted)** で失敗

```
openat(AT_FDCWD, "/sys/fs/bpf", O_RDONLY|O_DIRECTORY) = -1 EACCES (Permission denied)
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_SOCKET_FILTER, insn_cnt=2, insns=0x7fff839e1940, license="GPL", log_level=0, log_size=0, l
og_buf=NULL, kern_version=KERNEL_VERSION(0, 0, 0), prog_flags=0, prog_name="", prog_ifindex=0, expected_attach_type=BPF_CGROUP_INET_IN
GRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NULL, line_info_cnt=0, at
tach_btf_id=0, attach_prog_fd=0, fd_array=NULL}, 148) = -1 EPERM (Operation not permitted)
prlimit64(0, RLIMIT_MEMLOCK, {rlim_cur=RLIM64_INFINITY, rlim_max=RLIM64_INFINITY}, NULL) = -1 EPERM (Operation not permitted)
write(2, "libbpf: Failed to bump RLIMIT_ME"..., 90libbpf: Failed to bump RLIMIT_MEMLOCK (err = -EPERM), you might need to do it explic
itly!
) = 90
```

ftrace で sys_bpf の呼び出しをトレース
→ `bpf_prog_load()` で権限チェックが走る

- “`bpf_token_capable()`”
- “`ns_capable()`”

あたりが関係していそう 🤔

```
__sys_bpf() {
  bpf_check_uarg_tail_zero();
  __check_object_size() {
    check_stack_object();
  }
  security_bpf();
  bpf_prog_load() {
    bpf_token_capable() {
      ns_capable() {
        security_capable() {
          cap_capable();
        }
      }
      ns_capable() {
        security_capable() {
          cap_capable();
        }
      }
    }
    bpf_token_put();
  }
}
```

sudo trace-cmd record -p function_graph -g __sys_bpf sudo -u naoki ./sockex1 trace-cmd report –t > trace.log

スライド: https://pibvt.net/s/zUEhT3GJrUc

bpf_prog_load における権限チェック

```
2782        bpf_cap = bpf_token_capable(token, CAP_BPF);
2783        err = -EPERM;
2784
2785        if (!IS_ENABLED(CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS) &&
2786            (attr->prog_flags & BPF_F_ANY_ALIGNMENT) &&
2787            !bpf_cap)
2788                goto put_token;
2789
2790        /* Intent here is for unprivileged_bpf_disabled to block BPF program
2791         * creation for unprivileged users; other actions depend
2792         * on fd availability and access to bpffs, so are dependent on
2793         * object creation success. Even with unprivileged BPF disabled,
2794         * capability checks are still carried out for these
2795         * and other operations.
2796         */
2797        if (sysctl_unprivileged_bpf_disabled && !bpf_cap)
2798                goto put_token;
2799
2800        if (attr->insn_cnt == 0 ||
2801            attr->insn_cnt > (bpf_cap ? BPF_COMPLEXITY_LIMIT_INSNS : BPF_MAXINSNS)) {
2802                err = -E2BIG;
2803                goto put_token;
2804        }
2805        if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
2806            type != BPF_PROG_TYPE_CGROUP_SKB &&
2807            !bpf_cap)
2808                goto put_token;
2809
2810        if (is_net_admin_prog_type(type) && !bpf_token_capable(token, CAP_NET_ADMIN))
2811                goto put_token;
2812        if (is_perfmon_prog_type(type) && !bpf_token_capable(token, CAP_PERFMON))
2813                goto put_token;
```

“token” が CAP_BPF を持つかチェック

sysctl_unprivileged_bpf_disabled != “0”
かつ bpf_cap == NULL なら EPERM

一部の Program type は権限なしでも続行

Program type に応じて追加の権限チェック

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2740

スライド: https://pibvt.net/s/zUEhT3GJrUc

bpf_prog_load における権限チェック

```
2782  bpf_cap = bpf_token_capable(token, CAP_BPF);
2783  err = -EPERM;
2784
2785  if (!IS_ENABLED(CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS) &&
2786      (attr->prog_flags & BPF_F_ANY_ALIGNMENT) &&
2787      !bpf_cap)
2788          goto put_token;
2789
2790  /* Intent here is for unprivileged_bpf_disabled to block BPF program
2791   * creation for unprivileged users; other actions depend
2792   * on fd availability and access to bpffs, so are dependent on
2793   * object creation success. Even with unprivileged BPF disabled,
2794   * capability checks are still carried out for these
2795   * and other operations.
2796   */
2797  if (sysctl_unprivileged_bpf_disabled && !bpf_cap)
2798          goto put_token;
2799
2800  if (attr->insn_cnt == 0 ||
2801      attr->insn_cnt > (bpf_cap ? BPF_COMPLEXITY_LIMIT_INSNS : BPF_MAXINSNS)) {
2802          err = -E2BIG;
2803          goto put_token;
2804  }
2805  if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
2806      type != BPF_PROG_TYPE_CGROUP_SKB &&
2807      !bpf_cap)
2808          goto put_token;
2809
2810  if (is_net_admin_prog_type(type) && !bpf_token_capable(token, CAP_NET_ADMIN))
2811          goto put_token;
2812  if (is_perfmon_prog_type(type) && !bpf_token_capable(token, CAP_PERFMON))
2813          goto put_token;
```

“token” が CAP_BPF を持つかチェック

sysctl_unprivileged_bpf_disabled != “0”
かつ bpf_cap == NULL なら EPERM

一部の Program type は権限なしでも続行

Program type に応じて追加の権限チェック

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2740

スライド: https://pibvt.net/s/zUEhT3GJrUc

bpf_token_capable における処理

bpf_token の有無により分岐

- bpf_token 無し: ホストの UserNS(**init_user_ns**) で capability をチェック
- bpf_token 有り: **bpf_token** に紐づく UserNS で capability をチェック

※ CAP_SYS_ADMIN が付与されている場合は該当 capability がなくともパスする

```
bool bpf_token_capable(const struct bpf_token *token, int cap)
{
        struct user_namespace *userns;

        /* BPF token allows ns_capable() level of capabilities */
        userns = token ? token->userns : &init_user_ns;
        if (!bpf_ns_capable(userns, cap))
                return false;
        if (token && security_bpf_token_capable(token, cap) < 0)
                return false;
        return true;
}

static bool bpf_ns_capable(struct user_namespace *ns, int cap)
{
        return ns_capable(ns, cap) || (cap != CAP_SYS_ADMIN && ns_capable(ns, CAP_SYS_ADMIN));
}
```

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/token.c#L16

スライド: https://pibvt.net/s/zUEhT3GJrUc

bpf_prog_load における権限チェック

```
2782    bpf_cap = bpf_token_capable(token, CAP_BPF);
2783    err = -EPERM;
2784
2785    if (!IS_ENABLED(CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS) &&
2786        (attr->prog_flags & BPF_F_ANY_ALIGNMENT) &&
2787        !bpf_cap)
2788            goto put_token;
2789
2790    /* Intent here is for unprivileged_bpf_disabled to block BPF program
2791     * creation for unprivileged users; other actions depend
2792     * on fd availability and access to bpffs, so are dependent on
2793     * object creation success. Even with unprivileged BPF disabled,
2794     * capability checks are still carried out for these
2795     * and other operations.
2796     */
2797    if (sysctl_unprivileged_bpf_disabled && !bpf_cap)
2798            goto put_token;
2799
2800    if (attr->insn_cnt == 0 ||
2801        attr->insn_cnt > (bpf_cap ? BPF_COMPLEXITY_LIMIT_INSNS : BPF_MAXINSNS)) {
2802            err = -E2BIG;
2803            goto put_token;
2804    }
2805    if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
2806        type != BPF_PROG_TYPE_CGROUP_SKB &&
2807        !bpf_cap)
2808            goto put_token;
2809
2810    if (is_net_admin_prog_type(type) && !bpf_token_capable(token, CAP_NET_ADMIN))
2811            goto put_token;
2812    if (is_perfmon_prog_type(type) && !bpf_token_capable(token, CAP_PERFMON))
2813            goto put_token;
```

“token” が CAP_BPF を持つかチェック

sysctl_unprivileged_bpf_disabled != “0”
かつ bpf_cap == NULL なら EPERM

一部の Program type は権限なしでも続行

Program type に応じて追加の権限チェック

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2740

スライド: https://pibvt.net/s/zUEhT3GJrUc

unprivileged_bpf_disabled について

非特権ユーザによる bpf(2) の呼び出しを制限する(0: 許可, 1,2: 拒否)

背景: **eBPF の安全性に対する懸念**(CVE-2020-8835 等)
→ 各種ディストリビューションで呼び出し拒否をデフォルトに

- https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047
- https://www.suse.com/ja-jp/support/kb/doc/?id=000020545

unprivileged_bpf_disabled

Writing 1 to this entry will disable unprivileged calls to `bpf()`; once disabled, calling `bpf()` without `CAP_SYS_ADMIN` or `CAP_BPF` will return `-EPERM`. Once set to 1, this can't be cleared from the running kernel anymore.

Writing 2 to this entry will also disable unprivileged calls to `bpf()`, however, an admin can still change this setting later on, if needed, by writing 0 or 1 to this entry.

If `BPF_UNPRIV_DEFAULT_OFF` is enabled in the kernel config, then this entry will default to 2 instead of 0.

0	Unprivileged calls to `bpf()` are enabled
1	Unprivileged calls to `bpf()` are disabled without recovery
2	Unprivileged calls to `bpf()` are disabled

https://docs.kernel.org/admin-guide/sysctl/kernel.html#unprivileged-bpf-disabled

スライド: https://pibvt.net/s/zUEhT3GJrUc

bpf_prog_load における権限チェック

```
2782    bpf_cap = bpf_token_capable(token, CAP_BPF);
2783    err = -EPERM;
2784
2785    if (!IS_ENABLED(CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS) &&
2786        (attr->prog_flags & BPF_F_ANY_ALIGNMENT) &&
2787        !bpf_cap)
2788            goto put_token;
2789
2790    /* Intent here is for unprivileged_bpf_disabled to block BPF program
2791     * creation for unprivileged users; other actions depend
2792     * on fd availability and access to bpffs, so are dependent on
2793     * object creation success. Even with unprivileged BPF disabled,
2794     * capability checks are still carried out for these
2795     * and other operations.
2796     */
2797    if (sysctl_unprivileged_bpf_disabled && !bpf_cap)
2798            goto put_token;
2799
2800    if (attr->insn_cnt == 0 ||
2801        attr->insn_cnt > (bpf_cap ? BPF_COMPLEXITY_LIMIT_INSNS : BPF_MAXINSNS)) {
2802            err = -E2BIG;
2803            goto put_token;
2804    }
2805    if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
2806        type != BPF_PROG_TYPE_CGROUP_SKB &&
2807        !bpf_cap)
2808            goto put_token;
2809
2810    if (is_net_admin_prog_type(type) && !bpf_token_capable(token, CAP_NET_ADMIN))
2811            goto put_token;
2812    if (is_perfmon_prog_type(type) && !bpf_token_capable(token, CAP_PERFMON))
2813            goto put_token;
```

“token” が CAP_BPF を持つかチェック

sysctl_unprivileged_bpf_disabled != “0”
かつ bpf_cap == NULL なら EPERM

一部の Program type は権限なしでも続行

Program type に応じて追加の権限チェック

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2740

スライド: https://pibvt.net/s/zUEhT3GJrUc

Program type ごとの要求する権限

Program type に応じて追加 capability を要求

- XDP, SK_*, SCHED_*, LWT_* 等のネットワーク系
  → **CAP_NET_ADMIN**
- KPROBE, TRACEPOINT 等のトレーシング系
  → **CAP_PERFMON**

```c
static bool is_perfmon_prog_type(enum bpf_prog_type prog_type)
{
        switch (prog_type) {
        case BPF_PROG_TYPE_KPROBE:
        case BPF_PROG_TYPE_TRACEPOINT:
        case BPF_PROG_TYPE_PERF_EVENT:
        case BPF_PROG_TYPE_RAW_TRACEPOINT:
        case BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE:
        case BPF_PROG_TYPE_TRACING:
        case BPF_PROG_TYPE_LSM:
        case BPF_PROG_TYPE_STRUCT_OPS: /* has access to struct sock */
        case BPF_PROG_TYPE_EXT: /* extends any prog */
                return true;
        default:
                return false;
        }
}
```

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2719

```c
static bool is_net_admin_prog_type(enum bpf_prog_type prog_type)
{
        switch (prog_type) {
        case BPF_PROG_TYPE_SCHED_CLS:
        case BPF_PROG_TYPE_SCHED_ACT:
        case BPF_PROG_TYPE_XDP:
        case BPF_PROG_TYPE_LWT_IN:
        case BPF_PROG_TYPE_LWT_OUT:
        case BPF_PROG_TYPE_LWT_XMIT:
        case BPF_PROG_TYPE_LWT_SEG6LOCAL:
        case BPF_PROG_TYPE_SK_SKB:
        case BPF_PROG_TYPE_SK_MSG:
        case BPF_PROG_TYPE_FLOW_DISSECTOR:
        case BPF_PROG_TYPE_CGROUP_DEVICE:
        case BPF_PROG_TYPE_CGROUP_SOCK:
        case BPF_PROG_TYPE_CGROUP_SOCK_ADDR:
        case BPF_PROG_TYPE_CGROUP_SOCKOPT:
        case BPF_PROG_TYPE_CGROUP_SYSCTL:
        case BPF_PROG_TYPE_SOCK_OPS:
        case BPF_PROG_TYPE_EXT: /* extends any prog */
        case BPF_PROG_TYPE_NETFILTER:
                return true;
        case BPF_PROG_TYPE_CGROUP_SKB:
                /* always unpriv */
        case BPF_PROG_TYPE_SK_REUSEPORT:
                /* equivalent to SOCKET_FILTER. need CAP_BPF only */
        default:
                return false;
        }
}
```

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2688

スライド: https://pibvt.net/s/zUEhT3GJrUc

unprivileged_bpf_disabled=0 な場合

PROG_TYPE_SOCKET_FILTER は追加権限を要求しない

- CAP_BPF 無しでも許可される特殊な PROG_TYPE
- CAP_NET_ADMIN, CAP_PERFMON も要求されない

→ 非特権プロセスでも利用可能

※ CAP_NET_RAW は Raw socket 用

```
2805  if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
2806      type != BPF_PROG_TYPE_CGROUP_SKB &&
2807      !bpf_cap)
2808          goto put_token;
2809
2810  if (is_net_admin_prog_type(type) && !bpf_token_capable(token, CAP_NET_ADMIN))
2811          goto put_token;
2812  if (is_perfmon_prog_type(type) && !bpf_token_capable(token, CAP_PERFMON))
2813          goto put_token;
```

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L2740

```
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ echo 0 | sudo tee /proc/sys/kernel/unprivileged_bpf_disabled
0
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_net_raw+ep ./sockex1
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./sockex1
TCP 0 UDP 0 ICMP 0 bytes
TCP 0 UDP 0 ICMP 196 bytes
TCP 0 UDP 0 ICMP 392 bytes
TCP 0 UDP 0 ICMP 588 bytes
TCP 0 UDP 0 ICMP 784 bytes
```

スライド: https://pibvt.net/s/zUEhT3GJrUc

CAP_BPF を付与する場合

`unprivileged_bpf_disabled = 1 or 2` →そのままでは利用不可

CAP_BPF を付与すれば非特権プロセスでも利用可能

```
 130 naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ echo 2 | sudo tee /proc/sys/kernel/unprivileged_bpf_disabled
2
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./sockex1
libbpf: Failed to bump RLIMIT_MEMLOCK (err = -EPERM), you might need to do it explicitly!
libbpf: Error in bpf_object__probe_loading(): -EPERM. Couldn't load trivial BPF program. Make sure your kernel supports
BPF (CONFIG_BPF_SYSCALL=y) and/or that RLIMIT_MEMLOCK is set to big enough value.
libbpf: failed to load object './sockex1_kern.o'
 1 naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf,cap_net_raw+ep ./sockex1
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./sockex1
TCP 0 UDP 0 ICMP 0 bytes
TCP 0 UDP 0 ICMP 196 bytes
TCP 0 UDP 0 ICMP 392 bytes
TCP 0 UDP 0 ICMP 588 bytes
TCP 0 UDP 0 ICMP 784 bytes
```

スライド: https://pibvt.net/s/zUEhT3GJrUc

Map の作成に要する権限

基本的な Map は CAP_BPF で作成可能

※ 種類によっては追加の権限が必要

今回は “BPF_MAP_TYPE_ARRAY” を利用

→ CAP_BPF だけで動作する

```
struct {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, __u32);
        __type(value, long);
        __uint(max_entries, 256);
} my_map SEC(".maps");

SEC("socket1")
int bpf_prog1(struct __sk_buff *skb)
{
        int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
        long *value;
```

https://elixir.bootlin.com/linux/v6.14.6/source/samples/bpf/sockex1_kern.c

```
if (sysctl_unprivileged_bpf_disabled && !bpf_token_capable(token, CAP_BPF))
        goto put_token;

/* check privileged map type permissions */
switch (map_type) {
case BPF_MAP_TYPE_ARRAY:
case BPF_MAP_TYPE_PERCPU_ARRAY:
case BPF_MAP_TYPE_PROG_ARRAY:
case BPF_MAP_TYPE_PERF_EVENT_ARRAY:
case BPF_MAP_TYPE_CGROUP_ARRAY:
case BPF_MAP_TYPE_ARRAY_OF_MAPS:
case BPF_MAP_TYPE_HASH:
case BPF_MAP_TYPE_PERCPU_HASH:
case BPF_MAP_TYPE_HASH_OF_MAPS:
case BPF_MAP_TYPE_RINGBUF:
case BPF_MAP_TYPE_USER_RINGBUF:
case BPF_MAP_TYPE_CGROUP_STORAGE:
case BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE:
        /* unprivileged */
        break;
case BPF_MAP_TYPE_SK_STORAGE:
case BPF_MAP_TYPE_INODE_STORAGE:
case BPF_MAP_TYPE_TASK_STORAGE:
case BPF_MAP_TYPE_CGRP_STORAGE:
case BPF_MAP_TYPE_BLOOM_FILTER:
case BPF_MAP_TYPE_LPM_TRIE:
case BPF_MAP_TYPE_REUSEPORT_SOCKARRAY:
case BPF_MAP_TYPE_STACK_TRACE:
case BPF_MAP_TYPE_QUEUE:
case BPF_MAP_TYPE_STACK:
case BPF_MAP_TYPE_LRU_HASH:
case BPF_MAP_TYPE_LRU_PERCPU_HASH:
case BPF_MAP_TYPE_STRUCT_OPS:
case BPF_MAP_TYPE_CPUMAP:
case BPF_MAP_TYPE_ARENA:
        if (!bpf_token_capable(token, CAP_BPF))
                goto put_token;
        break;
case BPF_MAP_TYPE_SOCKMAP:
case BPF_MAP_TYPE_SOCKHASH:
case BPF_MAP_TYPE_DEVMAP:
case BPF_MAP_TYPE_DEVMAP_HASH:
case BPF_MAP_TYPE_XSKMAP:
        if (!bpf_token_capable(token, CAP_NET_ADMIN))
                goto put_token;
        break;
default:
        WARN(1, "unsupported map type %d", map_type);
        goto put_token;
}
```

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/syscall.c#L1318

スライド: https://pibvt.net/s/zUEhT3GJrUc

Map の操作に必要な権限

Map の操作に関しては CAP_BPF は必要ない(はず)

操作: BPF_MAP_{LOOKUP, UPDATE, DELETE}_ELEM

※ Map の取得方法次第では追加権限が必要な場合がある

BPF_GET_OBJ を使う場合: **BPF FS に pin されたファイルの RW 権限は必要**

→ **ほかのプロセスが操作できる**ため、MountNS を切り分離等すべき

```
naoki@ebpf-meetup:~/ebpf-meetup/map$ sudo ./creator
Map created successfully (fd: 3)
Map pinned to /sys/fs/bpf/my_pinned_map
Added initial element: key=1, value=12345
naoki@ebpf-meetup:~/ebpf-meetup/map$ sudo chmod 755 /sys/fs/bpf
naoki@ebpf-meetup:~/ebpf-meetup/map$ sudo chmod 666 /sys/fs/bpf/my_pinned_map
naoki@ebpf-meetup:~/ebpf-meetup/map$ ls -l /sys/fs/bpf/my_pinned_map
-rw-rw-rw- 1 root root 0 May 26 12:45 /sys/fs/bpf/my_pinned_map
naoki@ebpf-meetup:~/ebpf-meetup/map$ ./reader
Opened pinned map from /sys/fs/bpf/my_pinned_map (fd: 3)
Lookup for key 1: Found value 12345
Updating map with key 2, value 56789...
Successfully updated map for key 2.
Lookup for key 2: Found value 56789
Lookup for key 99: Key not found (as expected)
```

eBPF の活用領域

eBPF の活用領域は多岐にわたる

- **ネットワーク系**
  - Socket: Raw socket でのフィルタリング等
  - XDP, SKB, TC でのパケットのフィルタリングや改変
  - Lightweight Tunnel: トンネリングに関するフレームワーク
- **トレーシング系**
  - Kprobe, Tracepoint: カーネル内部の挙動をトレース
  - Uprobe: ユーザープロセスの挙動をトレース
- cgroup: cgroup 単位で SKB や Socket の挙動を制御

参考: https://docs.ebpf.io/linux/program-type/
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(XDP)

XDP は CAP_BPF+CAP_NET_ADMIN で動くはず
→そうとは限らない？

```
 130 naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf,cap_net_admin+ep ./xdp_adjust_tail
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./xdp_adjust_tail -S -i lo
libbpf: prog '_xdp_icmp': BPF program load failed: -EACCES
libbpf: prog '_xdp_icmp': -- BEGIN PROG LOAD LOG --
0: R1=ctx() R10=fp0
; int _xdp_icmp(struct xdp_md *xdp) @ xdp_adjust_tail_kern.c:138
0: (b4) w0 = 1                        ; R0_w=1
; void *data_end = (void *)(long)xdp->data_end; @ xdp_adjust_tail_kern.c:140
1: (61) r3 = *(u32 *)(r1 +4)          ; R1=ctx() R3_w=pkt_end()
; void *data = (void *)(long)xdp->data; @ xdp_adjust_tail_kern.c:141
2: (61) r2 = *(u32 *)(r1 +0)          ; R1=ctx() R2_w=pkt(r=0)
; if (eth + 1 > data_end) @ xdp_adjust_tail_kern.c:145
3: (bf) r4 = r2                       ; R2_w=pkt(r=0) R4_w=pkt(r=0)
4: (07) r4 += 14                      ; R4_w=pkt(off=14,r=0)
5: (2d) if r4 > r3 goto pc+121        ; R3_w=pkt_end() R4_w=pkt(off=14,r=14)
; h_proto = eth->h_proto; @ xdp_adjust_tail_kern.c:148
6: (71) r4 = *(u8 *)(r2 +12)          ; R2_w=pkt(r=14) R4_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff))
7: (71) r5 = *(u8 *)(r2 +13)          ; R2_w=pkt(r=14) R5_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff))
8: (64) w5 <<= 8                      ; R5_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=0xff00,var_off=(0x0; 0xff00))
9: (4c) w5 |= w4                      ; R4_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff)) R5_w=scalar(smin=s
min32=0,smax=umax=smax32=umax32=0xffff,var_off=(0x0; 0xffff))
10: (b4) w0 = 2                       ; R0_w=2
; if (h_proto == htons(ETH_P_IP)) @ xdp_adjust_tail_kern.c:150
11: (56) if w5 != 0x8 goto pc+115     ; R5_w=8
; int pckt_size = data_end - data; @ xdp_adjust_tail_kern.c:125
12: (1c) w3 -= w2
R3 pointer -= pointer prohibited
processed 13 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0 mark_read 0
-- END PROG LOAD LOG --
libbpf: prog '_xdp_icmp': failed to load: -EACCES
libbpf: failed to load object './xdp_adjust_tail_kern.o'
```

https://elixir.bootlin.com/linux/v6.14.6/source/samples/bpf/xdp_adjust_tail_kern.c

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(XDP)

特権 or **CAP_BPF+CAP_NET_ADMIN+CAP_PERFMON** なら動く

```
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo ./xdp_adjust_tail -S -i lo
icmp "packet too big" sent:          0 pkts
^Cnaoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf,cap_net_admin,cap_perfmon+ep ./xdp_adjust_tail
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./xdp_adjust_tail -S -i lo
icmp "packet too big" sent:          0 pkts
```

Verifier の挙動が capability に応じて異なる場合がある

```
8: (64) w5 <<= 8                      ; R5_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=0xff00,var_off=(0x0; 0xff00))
9: (4c) w5 |= w4                      ; R4_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff)) R5_w=scalar(smin=s
min32=0,smax=umax=smax32=umax32=0xffff,var_off=(0x0; 0xffff))
10: (b4) w0 = 2                       ; R0_w=2
; if (h_proto == htons(ETH_P_IP)) @ xdp_adjust_tail_kern.c:150
11: (56) if w5 != 0x8 goto pc+115     ; R5_w=8
; int pckt_size = data_end - data; @ xdp_adjust_tail_kern.c:125
12: (1c) w3 -= w2
R3 pointer -= pointer prohibited
processed 13 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0 mark_read 0
-- END PROG LOAD LOG --
libbpf: prog '_xdp_icmp': failed to load: -EACCES
libbpf: failed to load object './xdp_adjust_tail_kern.o'
```

スライド: https://pibvt.net/s/zUEhT3GJrUc

CAP_PERFMON の有無による挙動の違い

Verifier 内では検査において一定の緩和を許すオプションが存在

- Specter 関連のチェック
  参考: https://mmi.hatenablog.com/entry/2018/02/02/003325
- ポインタ, スタック関連のチェック

CAP_PERFMON を持つ場合、すべて無効化

= Verifier が緩い検査を行う

→ 厳密な検査を行っていない場合がある

```
env->allow_ptr_leaks = bpf_allow_ptr_leaks(env->prog->aux->token);
env->allow_uninit_stack = bpf_allow_uninit_stack(env->prog->aux->token);
env->bypass_spec_v1 = bpf_bypass_spec_v1(env->prog->aux->token);
env->bypass_spec_v4 = bpf_bypass_spec_v4(env->prog->aux->token);
env->bpf_capable = is_priv = bpf_token_capable(env->prog->aux->token, CAP_BPF);
```

https://elixir.bootlin.com/linux/v6.14.6/source/kernel/bpf/verifier.c#L23110

```
static inline bool bpf_allow_ptr_leaks(const struct bpf_token *token)
{
        return bpf_token_capable(token, CAP_PERFMON);
}

static inline bool bpf_allow_uninit_stack(const struct bpf_token *token)
{
        return bpf_token_capable(token, CAP_PERFMON);
}

static inline bool bpf_bypass_spec_v1(const struct bpf_token *token)
{
        return cpu_mitigations_off() || bpf_token_capable(token, CAP_PERFMON);
}

static inline bool bpf_bypass_spec_v4(const struct bpf_token *token)
{
        return cpu_mitigations_off() || bpf_token_capable(token, CAP_PERFMON);
}
```

https://elixir.bootlin.com/linux/v6.14.6/source/include/linux/bpf.h#L2409

スライド: https://pibvt.net/s/zUEhT3GJrUc

CAP_PERFMON の有無による挙動の違い

安全性への懸念

- CAP_PERFMON:
  カーネルのメモリを読む必要があるため、
  意図的に緩和している
- CAP_NET_ADMIN:
  脆弱性への対応を含む厳密なチェックを行う

→ Verifier における検査の違いも含め、
必要最低限な capability の付与が重要

```
commit 2c78ee898d8f10ae6fb2fa23a3fbaec96b1b7366
Author: Alexei Starovoitov <ast@kernel.org>
Date:   Wed May 13 16:03:54 2020 -0700

    bpf: Implement CAP_BPF

    Implement permissions as stated in uapi/linux/capability.h
    In order to do that the verifier allow_ptr_leaks flag is split
    into four flags and they are set as:
      env->allow_ptr_leaks = bpf_allow_ptr_leaks();
      env->bypass_spec_v1 = bpf_bypass_spec_v1();
      env->bypass_spec_v4 = bpf_bypass_spec_v4();
      env->bpf_capable = bpf_capable();

    The first three currently equivalent to perfmon_capable(), since leaking kernel
    pointers and reading kernel memory via side channel attacks is roughly
    equivalent to reading kernel memory with cap_perfmon.

    'bpf_capable' enables bounded loops, precision tracking, bpf to bpf calls and
    other verifier features. 'allow_ptr_leaks' enable ptr leaks, ptr conversions,
    subtraction of pointers. 'bypass_spec_v1' disables speculative analysis in the
    verifier, run time mitigations in bpf array, and enables indirect variable
    access in bpf programs. 'bypass_spec_v4' disables emission of sanitation code
    by the verifier.

    That means that the networking BPF program loaded with CAP_BPF + CAP_NET_ADMIN
    will have speculative checks done by the verifier and other spectre mitigation
    applied. Such networking BPF program will not be able to leak kernel pointers
    and will not be able to access arbitrary kernel memory.

    Signed-off-by: Alexei Starovoitov <ast@kernel.org>
    Signed-off-by: Daniel Borkmann <daniel@iogearbox.net>
    Link: https://lore.kernel.org/bpf/20200513230355.7858-3-alexei.starovoitov@gmail
```

https://lore.kernel.org/bpf/20200513230355.7858-3-alexei.starovoitov@gmail.com/

スライド: https://pibvt.net/s/zUEhT3GJrUc

iproute2 によるロードの場合

sudo ip link set… で XDP プログラムをロードした場合、エラーとならない

capsh(1) で CAP_PERFMON,CAP_SYS_ADMIN を drop した場合、Verifier でエラーとなる

```
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo capsh --drop=cap_perfmon,cap_sys_admin -- -bash -c "ip link set dev lo xdpgeneric obj xdp_adjust_tail_kern.o sec xdp_icmp"
libbpf: prog '_xdp_icmp': BPF program load failed: Permission denied
libbpf: prog '_xdp_icmp': -- BEGIN PROG LOAD LOG --
0: R1=ctx() R10=fp0
; int _xdp_icmp(struct xdp_md *xdp) @ xdp_adjust_tail_kern.c:138
0: (b4) w0 = 1                        ; R0_w=1
; void *data_end = (void *)(long)xdp->data_end; @ xdp_adjust_tail_kern.c:140
1: (61) r3 = *(u32 *)(r1 +4)          ; R1=ctx() R3_w=pkt_end()
; void *data = (void *)(long)xdp->data; @ xdp_adjust_tail_kern.c:141
2: (61) r2 = *(u32 *)(r1 +0)          ; R1=ctx() R2_w=pkt(r=0)
; if (eth + 1 > data_end) @ xdp_adjust_tail_kern.c:145
3: (bf) r4 = r2                       ; R2_w=pkt(r=0) R4_w=pkt(r=0)
4: (07) r4 += 14                      ; R4_w=pkt(off=14,r=0)
5: (2d) if r4 > r3 goto pc+121        ; R3_w=pkt_end() R4_w=pkt(off=14,r=14)
; h_proto = eth->h_proto; @ xdp_adjust_tail_kern.c:148
6: (71) r4 = *(u8 *)(r2 +12)          ; R2_w=pkt(r=14) R4_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff))
7: (71) r5 = *(u8 *)(r2 +13)          ; R2_w=pkt(r=14) R5_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff))
8: (64) w5 <<= 8                      ; R5_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=0xff00,var_off=(0x0; 0xff00))
9: (4c) w5 |= w4                      ; R4_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=255,var_off=(0x0; 0xff)) R5_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=0xffff,var_off=(0x0; 0xffff))
10: (b4) w0 = 2                       ; R0_w=2
; if (h_proto == htons(ETH_P_IP)) @ xdp_adjust_tail_kern.c:150
11: (56) if w5 != 0x8 goto pc+115     ; R5_w=8
; int pckt_size = data_end - data; @ xdp_adjust_tail_kern.c:125
12: (1c) w3 -= w2
R3 pointer -= pointer prohibited
processed 13 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0 mark_read 0
-- END PROG LOAD LOG --
libbpf: prog '_xdp_icmp': failed to load: -13
libbpf: failed to load object 'xdp_adjust_tail_kern.o'
 255 naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo ip link set dev lo xdpgeneric obj xdp_adjust_tail_kern.o sec xdp_icmp
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo ip link set dev lo xdpgeneric off
```

eBPF の活用領域

eBPF の活用領域は多岐にわたる

- **ネットワーク系**
  - Socket: Raw socket でのフィルタリング等
  - XDP, SKB, TC でのパケットのフィルタリングや改変
  - Lightweight Tunnel: トンネリングに関するフレームワーク
- **トレーシング系**
  - Kprobe, Tracepoint: カーネル内部の挙動をトレース
  - Uprobe: ユーザープロセスの挙動をトレース
- cgroup: cgroup 単位で SKB や Socket の挙動を制御

参考: https://docs.ebpf.io/linux/program-type/
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(Kprobe)

Kprobe については

CAP_BPF+CAP_PERFMON で利用可能

Map についても **PERF_EVENT** 等が利用可能

```
struct {
        __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
        __uint(key_size, sizeof(int));
        __uint(value_size, sizeof(u32));
        __uint(max_entries, 2);
} my_map SEC(".maps");
```

```
SEC("ksyscall/write")
int bpf_prog1(struct pt_regs *ctx)
{
        struct S {
                u64 pid;
                u64 cookie;
        } data;

        data.pid = bpf_get_current_pid_tgid();
        data.cookie = 0x12345678;

        bpf_perf_event_output(ctx, &my_map, 0, &data, sizeof(data));

        return 0;
}
```

```
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf,cap_perfmon+ep ./trace_output
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./trace_output
Program: bpf_prog1, Type: 2
recv 14994 events per sec
99989+0 records in
99989+0 records out
51194368 bytes (51 MB, 49 MiB) copied, 6.66541 s, 7.7 MB/s
```

https://elixir.bootlin.com/linux/v6.14.6/source/samples/bpf/trace_output.bpf.c

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(Tracepoint)

Tracepoint について **CAP_BPF+CAP_PERFMON** では動作しない場合がある
→ また権限が足りない？

```
 130 naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./trace_output
libbpf: Failed to bump RLIMIT_MEMLOCK (err = -EPERM), you might need to do it explicitly!
libbpf: Error in bpf_object__probe_loading(): -EPERM. Couldn't load trivial BPF program. Make sure your kernel supports BPF (CONFIG_B
PF_SYSCALL=y) and/or that RLIMIT_MEMLOCK is set to big enough value.
libbpf: failed to load object './trace_output.bpf.o'
ERROR: loading BPF object file failed
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf+ep ./trace_output
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./trace_output
libbpf: prog 'bpf_prog1': BPF program load failed: -EPERM
libbpf: prog 'bpf_prog1': failed to load: -EPERM
libbpf: failed to load object './trace_output.bpf.o'
ERROR: loading BPF object file failed
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf,cap_perfmon+ep ./trace_output
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./trace_output
Program: bpf_prog1, Type: 5
libbpf: failed to determine tracepoint 'syscalls/sys_enter_write' perf event ID: -EACCES
libbpf: prog 'bpf_prog1': failed to create tracepoint 'syscalls/sys_enter_write' perf event: -EACCES
ERROR: bpf_program__attach failed
```

https://elixir.bootlin.com/linux/v6.14.6/source/samples/bpf/trace_output.bpf.c を元に
SEC(“tracepoint/syscalls/sys_enter_write”) に改変

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(Tracepoint)

`BPF_PROG_LOAD` については成功

`/sys/kernel/tracing/events/syscalls/sys_enter_write/id` の `openat(2)` に失敗

Tracepoint に関する eBPF プログラムアタッチの流れ

1. eBPF プログラムのロード
2. 対応する tracing point について id を取得し `perf_event_open(2)` で準備 ←**これに失敗**
3. eBPF プログラムと perf_event を `BPF_LINK_CREATE` でリンク

```
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=15, insns=0x5ed84571ca80, license="GPL", log_level=0, log_siz
e=0, log_buf=NULL, kern_version=KERNEL_VERSION(6, 14, 6), prog_flags=0, prog_name="bpf_prog1", prog_ifindex=0, expected_attach
_type=BPF_CGROUP_INET_INGRESS, prog_btf_fd=3, func_info_rec_size=8, func_info=0x5ed84571cb00, func_info_cnt=1, line_info_rec_s
ize=16, line_info=0x5ed84571cbf0, line_info_cnt=6, attach_btf_id=0, attach_prog_fd=0, fd_array=NULL}, 152) = 5
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0x1), ...}) = 0
write(1, "Program: bpf_prog1, Type: 5\n", 28) = 28
faccessat2(AT_FDCWD, "/sys/kernel/debug/tracing", F_OK, AT_EACCESS) = -1 EACCES (Permission denied)
openat(AT_FDCWD, "/sys/kernel/tracing/events/syscalls/sys_enter_write/id", O_RDONLY|O_CLOEXEC) = -1 EACCES (Permission denied)
write(2, "libbpf: failed to determine trac"..., 89libbpf: failed to determine tracepoint 'syscalls/sys_enter_write' perf event
 ID: -EACCES
) = 89
write(2, "libbpf: prog 'bpf_prog1': failed"..., 101libbpf: prog 'bpf_prog1': failed to create tracepoint 'syscalls/sys_enter_w
rite' perf event: -EACCES
```

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF プログラム動作の流れ(Tracepoint)

`/sys/kernel/tracing/events/syscalls/sys_enter_write/id` を
非特権プロセスで読めるようにすれば解決

解決策1: `CAP_DAC_READ_SEARCH`

- ディレクトリとファイルの読み出しおよび実行権限のチェックをバイパス
- (注!) 強力な権限であるため、扱いには細心の注意が必要

解決策2: `chmod(1)` で非特権プロセスが読めるように変更

解決策3: id を静的に与えて動作するように Loader を改良

```
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ sudo setcap cap_bpf,cap_perfmon,cap_dac_read_search+ep ./trace_output
naoki@ebpf-meetup:~/linux-6.14.6/samples/bpf$ ./trace_output
Program: bpf_prog1, Type: 5
recv 15154 events per sec
99985+0 records in
99985+0 records out
51192320 bytes (51 MB, 49 MiB) copied, 6.59508 s, 7.8 MB/s
```

eBPF の活用領域

eBPF の活用領域は多岐にわたる

- **ネットワーク系**
  - Socket: Raw socket でのフィルタリング等
  - XDP, SKB, TC でのパケットのフィルタリングや改変
  - Light Weight Tunnel: トンネリングに関するフレームワーク
- **トレーシング系**
  - Kprobe, Tracepoint: カーネル内部の挙動をトレース
  - Uprobe: ユーザープロセスの挙動をトレース
- cgroup: cgroup 単位で SKB や Socket の挙動を制御

参考: https://docs.ebpf.io/linux/program-type/
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

スライド: https://pibvt.net/s/zUEhT3GJrUc

Uprobe について

ユーザープロセスの関数呼び出し等をトレースする仕組み

- 特定の関数呼び出しをトレースして引数を読みだすことが可能
- OpenTelemetry の自動計装等でも活用されている

```c
// この関数をeBPFでトレースします
void my_target_function(int loop_count, const char *message) {
    printf("TargetApp (PID: %d): my_target_function called! Count: %d, Message: %s\n",
           getpid(), loop_count, message);
}

int main() {
    int i = 0;
    const char *messages[] = {"Hello", "eBPF", "Tracing", "Example", "Done"};

    printf("TargetApp (PID: %d): Starting up. Will call my_target_function periodically.\n", getpid()
```

↑この関数をトレースする

↓ eBPF プログラムで引数を記録

```c
SEC("uprobe/my_uprobe_program")
int trace_my_target_function_entry(struct pt_regs *ctx) {
    struct event_data data = {};
    u64 current_pid_tgid = bpf_get_current_pid_tgid();

    data.pid = current_pid_tgid >> 32; // 上位32ビットがPID

    // my_target_function(int loop_count, const char *message) の最初の引数を取得
    // x86_64では、関数の最初の整数/ポインタ引数はRDIレジスタに格納される
    // PT_REGS_PARM1(ctx) マクロでアクセス可能 (vmlinux.hが必要)
    data.first_arg = (s32)PT_REGS_PARM1(ctx);

    // perf eventをユーザー空間に送信
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &data, sizeof(data));

    return 0;
}
```

```c
// 4. uprobeをアタッチ
LIBBPF_OPTS(bpf_uprobe_opts, uprobe_options,
            .func_name = "my_target_function",
            .retprobe = false);

bpf_link = bpf_program__attach_uprobe_opts(
    bpf_prog,
    child_pid,
    "./target_app",
    0,
    &uprobe_options);
if (libbpf_get_error(bpf_link)) {
```

トレース対象の

- 実行バイナリ
- 関数名
- PID

を指定して eBPF プログラムをアタッチ

スライド: https://pibvt.net/s/zUEhT3GJrUc

eBPF 動作の流れ(Uprobe)

Uprobe は eBPF のプログラムとしては **TYPE_KPROBE** となる

→ **CAP_BPF+CAP_PERFMON** だけで動く

```
 1 naoki@ebpf-meetup:~/ebpf-meetup/uprobe$ sudo setcap cap_bpf+ep ./trace_custom_loader
naoki@ebpf-meetup:~/ebpf-meetup/uprobe$ ./trace_custom_loader
Loader: Target application './target_app' launched with PID: 41409
TargetApp (PID: 41409): Starting up. Will call my_target_function periodically.
TargetApp (PID: 41409): my_target_function called! Count: 0, Message: Hello
libbpf: prog 'trace_my_target_function_entry': BPF program load failed: Operation not permitted
libbpf: prog 'trace_my_target_function_entry': failed to load: -1
libbpf: failed to load object 'trace_custom_func.bpf.o'
Loader: Failed to load BPF object: Operation not permitted
Loader: Cleaning up...
Loader: Ensuring target application (PID 41409) is terminated.
Loader: Exited.
 1 naoki@ebpf-meetup:~/ebpf-meetup/uprobe$ sudo setcap cap_bpf,cap_perfmon+ep ./trace_custom_loader
naoki@ebpf-meetup:~/ebpf-meetup/uprobe$ ./trace_custom_loader
Loader: Target application './target_app' launched with PID: 41420
TargetApp (PID: 41420): Starting up. Will call my_target_function periodically.
TargetApp (PID: 41420): my_target_function called! Count: 0, Message: Hello
Loader: BPF object loaded successfully.
Loader: Successfully attached uprobe to ./target_app:my_target_function (PID 41420).
Loader: Listening for eBPF events. Press Ctrl+C to exit.
TargetApp (PID: 41420): my_target_function called! Count: 1, Message: eBPF
Loader: Warning - Received data of unexpected size 12, expected 8
```

https://www.kernel.org/doc/html/v6.14/trace/uprobetracer.html

スライド: https://pibvt.net/s/zUEhT3GJrUc

必要な権限まとめ

1. SOCKET_FILTER (Socket)
   → **CAP_BPF**
2. ネットワーク系(XDP, SKB 等)
   → **CAP_BPF+CAP_NET_ADMIN**
3. トレーシング系(Kprobe, Tracepoint, Uprobe 等)
   → **CAP_BPF+CAP_PERFMON**
   ただし、Tracepoint は対応する id が必要

※ `unprivileged_bpf_disabled=0` な場合、`CAP_BPF` は必要ない

CAP_PERFMON では Verifier の検査が緩い場合がある

→ 対応するプロセスには必要最低限の capability のみを割り当てることが重要

非特権プロセスに付与する権限としては、まだ大きい(特にCAP_NET_ADMIN)

スライド: https://pibvt.net/s/zUEhT3GJrUc

蛇足: さらに必要な capability

`BPF_MAP_GET_FD_BY_ID` は `SYS_ADMIN` が必要

helper function を使う場合も追加の capability が必要なことも

→ トライアンドエラー+カーネルコードリーディングに近い形になる場合もある

Helper Function	Purpose	Required Minimum Capabilities
bpf_probe_write_user	Write to any process's user space memory	CAP_SYS_ADMIN (& kernel lockdown[4])
bpf_probe_read_user	Read any process's user space memory	CAP_BPF & CAP_PERFMON
bpf_override_return	Alter return code of a kernel function	CAP_SYS_ADMIN
bpf_send_signal	Send a signal to kill any process	CAP_SYS_ADMIN

https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20—%20eBPF%20Security%20Threat%20Model.pdf

スライド: https://pibvt.net/s/zUEhT3GJrUc

本発表の流れ

eBPF の利用に必要な権限とその関係は実は複雑

「**非特権プロセスや(Rootless)コンテナで eBPF は使えるのか？**」

目次

1. eBPF が動作するまでの流れと要求する権限
    - 各種 Program Type ごとに(Socket, XDP, Kprobe Tracepoint, Uprobe)
2. BPF Token による権限管理
3. (Rootless)コンテナにおける eBPF の活用例
4. まとめ

BPF Token

BPF Token: Linux Kernel v6.9 から導入された機能

Token ベースで**細粒度な権限管理**を実現する

- delegate_cmds: `bpf(2)` で行う操作
- delegate_maps: eBPF プログラムに含まれるマップの種類
- delegate_progs: eBPF プログラムに含まれるプログラムの種類
- delegate_attachs: eBPF プログラムのアタッチ先

  指定可能な項目は各種定義に従う
  https://elixir.bootlin.com/linux/v6.14.6/source/include/uapi/linux/bpf.h#L144

`fsconfig(2)` で BPF Token に対して権限付与を行う

※ 付与には `CAP_SYS_ADMIN` が必要

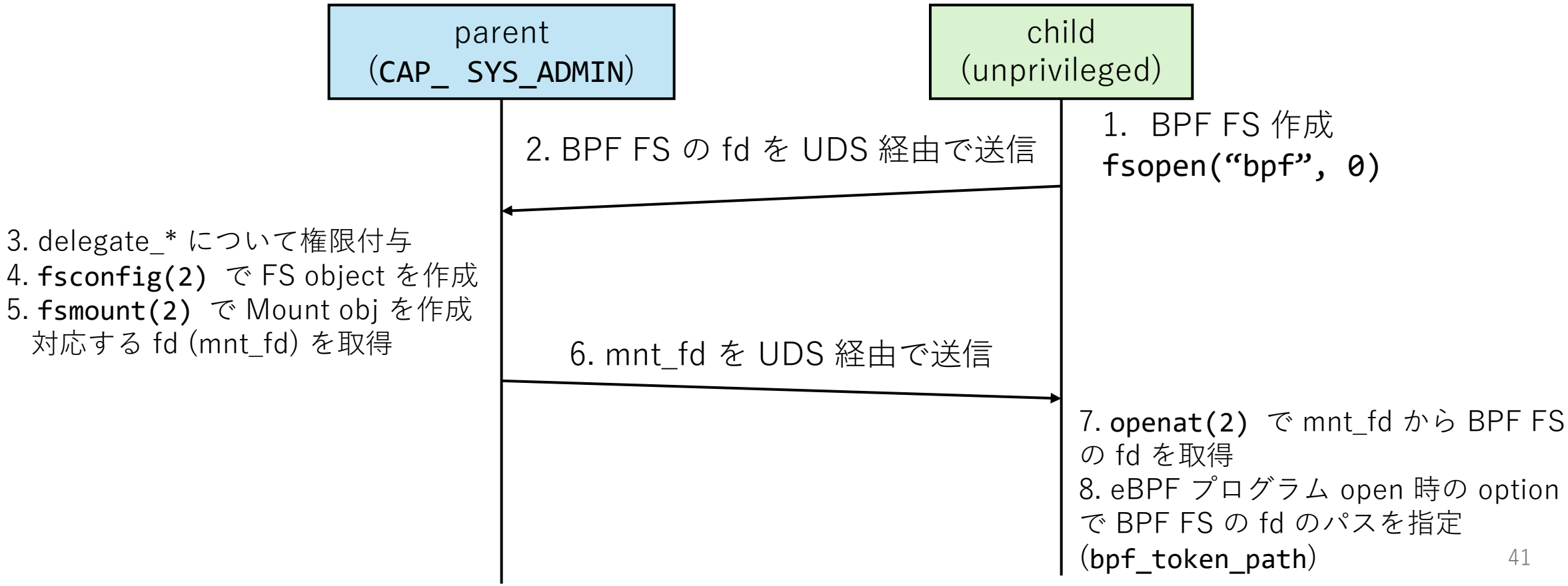
参考: https://patchwork.kernel.org/project/linux-fsdevel/cover/20230919214800.3803828-1-andrii@kernel.org/

スライド: https://pibvt.net/s/zUEhT3GJrUc

BPF Token の使い方

Linux Kernel のテストコードより

https://elixir.bootlin.com/linux/v6.14.6/source/tools/testing/selftests/bpf/prog_tests/token.c



スライド: https://pibvt.net/s/zUEhT3GJrUc

具体例(child)

```c
/* setup mountns to allow creating BPF FS (fsopen("bpf
err = unshare(CLONE_NEWUSER | CLONE_NEWNS);
if (err)
    goto cleanup;
printf("child: unshare\n");

fs_fd = create_bpffs_fd();
if (fs_fd < 0)
{
    printf("child: create_bpffs_fd failed\n");
    err = -EINVAL;
    goto cleanup;
}
printf("child: create_bpffs_fd\n");

/* pass BPF FS context object to parent */
err = sendfd(sock_fd, fs_fd);
if (err)
    goto cleanup;
zclose(fs_fd);
printf("child: sendfd\n");

/* avoid mucking around with mount namespaces and mount
 * well-known path, just get detach-mounted BPF FS fd
 */
err = recvfd(sock_fd, &mnt_fd);
if (err)
    goto cleanup;
printf("child: recvfd\n");

zclose(fs_fd);

bpffs_fd = openat(mnt_fd, ".", 0, O_RDWR);
if (bpffs_fd <= 0)
{
    err = -EINVAL;
    goto cleanup;
}
printf("child: bpffs_fd openat\n");

/* do custom test logic with customly set up BPF FS ins
err = callback(bpffs_fd);
if (err)
    goto cleanup;
```

新しい UserNS と MountNS へ移行
(BPF FS を非特権で作成するため)

BPF FS を作成
https://elixir.bootlin.com/linux/v6.14.6/source/tools/testing/selftests/bpf/prog_tests/token.c#L104

parent に BPF FS の fd を送信

parent から Mount obj の fd を受信

BPF FS fdを取得

```c
static int userns_obj_priv_map(int mnt_fd)
{
    printf("userns_obj_priv_map\n");
    LIBBPF_OPTS(bpf_object_open_opts, opts);
    char buf[256];
    struct priv_map *skel;
    int err;

    /* use bpf_token_path to provide BPF FS path */
    snprintf(buf, sizeof(buf), "/proc/self/fd/%d", mnt_fd);
    opts.bpf_token_path = buf;
    skel = priv_map__open_opts(&opts);
    if (!skel)
    {
        printf("failed priv_map__open_opts\n");
        return -EINVAL;
    }
    else
    {
        printf("succeeded priv_map__open_opts\n");
    }

    err = priv_map__load(skel);
    priv_map__destroy(skel);
    if (err)
        return -EINVAL;
    printf("succeeded priv_map__load\n");
    return 0;
}
```

BPF FS fd のパスを指定し open

eBPF プログラムをロード

スライド: https://pibvt.net/s/zUEhT3GJrUc

具体例(parent)

https://elixir.bootlin.com/linux/v6.14.6/source/tools/testing/selftests/bpf/prog_tests/token.c#L115

```c
err = recvfd(sock_fd, &fs_fd);
if (err)
    goto cleanup;
printf("parent: recvfd\n");

mnt_fd = materialize_bpffs_fd(fs_fd, bpffs_opts);
if (mnt_fd < 0)
{
    err = -EINVAL;
    goto cleanup;
}
printf("parent: materialize_bpffs_fd\n");
zclose(fs_fd);

/* pass BPF FS context object to parent */
err = sendfd(sock_fd, mnt_fd);
if (err)
    goto cleanup;
zclose(mnt_fd);
printf("parent: sendfd\n");
```

child から BPF FS fd を受信

fd に権限を付与

child へ Mount obj fd を送信

```c
static int materialize_bpffs_fd(int fs_fd, struct bpffs_opts *opts)
{
    int mnt_fd, err;

    /* set up token delegation mount options */
    err = set_delegate_mask(fs_fd, "delegate_cmds", opts->cmds, opts->cmds_str);
    if (err != 0)
        return err;
    err = set_delegate_mask(fs_fd, "delegate_maps", opts->maps, opts->maps_str);
    if (err != 0)
        return err;
    err = set_delegate_mask(fs_fd, "delegate_progs", opts->progs, opts->progs_str);
    if (err != 0)
        return err;
    err = set_delegate_mask(fs_fd, "delegate_attachs", opts->attachs, opts->attachs_str);
    if (err != 0)
        return err;

    /* instantiate FS object */
    err = sys_fsconfig(fs_fd, FSCONFIG_CMD_CREATE, NULL, NULL, 0);
    if (err < 0)
        return -errno;

    /* create O_PATH fd for detached mount */
    mnt_fd = sys_fsmount(fs_fd, 0, 0);
    if (err < 0)
        return -errno;

    return mnt_fd;
}
```

fd に権限を付与

BPF FS obj を作成

Mount obj を作成

スライド: https://pibvt.net/s/zUEhT3GJrUc

BPF Token の動作(許可)

BPF_MAP_TYPE_QUEUE を含む eBPF プログラムをロード↓

parent においては以下の付与権限を設定

```c
struct bpffs_opts opts = {
    .progs = bit(BPF_PROG_TYPE_SOCKET_FILTER),
    .cmds = bit(BPF_MAP_CREATE) | bit(BPF_PROG_LOAD),
    .maps = bit(BPF_MAP_TYPE_QUEUE),
    .attachs = ~0ULL, // 全てのattachを許可
};
```

```c
char _license[] SEC("license") = "GPL";

struct {
    __uint(type, BPF_MAP_TYPE_QUEUE);
    __uint(max_entries, 1);
    __type(value, __u32);
} priv_map SEC(".maps");
```

parent は CAP_SYS_ADMIN を要求

```
naoki@ebpf-meetup:~/ebpf-meetup/bpf-token-2$ sudo ./parent
[sudo] password for naoki:
accepted
parent: recvfd
parent: materialize_bpffs_fd
parent: sendfd
```

child は非特権で動作

```
naoki@ebpf-meetup:~/ebpf-meetup/bpf-token-2$ ./child2 map
connected
child: unshare
child: create_bpffs_fd
child: sendfd
child: recvfd
child: bpffs_fd openat
userns_obj_priv_map
succeeded priv_map__open_opts
succeeded priv_map__load
```

スライド: https://pibvt.net/s/zUEhT3GJrUc

BPF Token の動作(拒否)

BPF Token に付与された権限では足りない場合
→ BPF_PROG_LOAD で失敗 (EPERM)

```c
char _license[] SEC("license") = "GPL";

SEC("kprobe")
int kprobe_prog(void *ctx)
{
    return 1;
}
```

←の場合、↓の権限を要求

```c
struct bpffs_opts opts = {
    .progs = bit(BPF_PROG_TYPE_SOCKET_FILTER) | bit(BPF_PROG_TYPE_KPROBE),
    .cmds = bit(BPF_PROG_LOAD),
    .attachs = ~0ULL, // 全てのattachを許可
};
```

```
naoki@ebpf-meetup:~/ebpf-meetup/bpf-token-2$ sudo ./parent
[sudo] password for naoki:
accepted
parent: recvfd
parent: materialize_bpffs_fd
parent: sendfd
accepted
parent: recvfd
parent: materialize_bpffs_fd
parent: sendfd
```

```
naoki@ebpf-meetup:~/ebpf-meetup/bpf-token-2$ ./child2 prog
connected
child: unshare
child: create_bpffs_fd
child: sendfd
child: recvfd
child: bpffs_fd openat
succeeded priv_prog__open_opts
libbpf: prog 'kprobe_prog': BPF program load failed: Operation not pe
rmitted
libbpf: prog 'kprobe_prog': failed to load: -1
libbpf: failed to load object 'priv_prog'
libbpf: failed to load BPF skeleton 'priv_prog': -1
```

EPERM でロードに失敗

スライド: https://pibvt.net/s/zUEhT3GJrUc

本発表の流れ

eBPF の利用に必要な権限とその関係は実は複雑

「**非特権プロセスや(Rootless)コンテナで eBPF は使えるのか？**」

目次

1. eBPF が動作するまでの流れと要求する権限
   - 各種 Program Type ごとに(Socket, XDP, Kprobe, Tracepoint, Uprobe)
2. BPF Token による権限管理
3. (Rootless)コンテナにおける eBPF の活用例
4. まとめ

スライド: https://pibvt.net/s/zUEhT3GJrUc

Rootless コンテナにおける BPF Token

以下のいずれかの設定をすることで利用可能
(parent が作成した UDS を非特権プロセスで開く権限設定は必要)

1. UserNS, MountNS を作成するために **`unshare(2)`** を Seccomp で許可する
2. **`CAP_SYS_ADMIN`**(BPF FS 作成), **`CAP_BPF`**(BPF Token 作成)を追加する
   ※ **Rootless コンテナなら** UserNS が切られているため、リスク増加は限定的

1. unshare(2) 許可

```
 139 naoki@ebpf-meetup:~/ebpf-meetup/bpf-token-2$ nerdctl run -it -v .:/token --rm
  --security-opt seccomp=unconfined child2
root@938afa8f8df7:/# cd token/
root@938afa8f8df7:/token# ./child2 map
connected
child: unshare
child: create_bpffs_fd
child: sendfd
child: recvfd
child: sys_fspick
child: bpffs_fd openat
userns_obj_priv_map
succeeded priv_map__open_opts
succeeded priv_map__load
root@938afa8f8df7:/token#
```

2. capability 付与

```
naoki@ebpf-meetup:~/ebpf-meetup/bpf-token-2$ nerdctl run -it -v .:/token --rm --
cap-add CAP_SYS_ADMIN,CAP_BPF  child2
root@a66bd68eab0e:/# cd token/
root@a66bd68eab0e:/token# ./child2 map
connected
child: create_bpffs_fd
child: sendfd
child: recvfd
child: bpffs_fd openat
userns_obj_priv_map
succeeded priv_map__open_opts
succeeded priv_map__load
root@a66bd68eab0e:/token#
```

スライド: https://pibvt.net/s/zUEhT3GJrUc

コンテナにおける活用例

- Uprobe を利用するアプリ(OpenTelemetry の自動計装等)の安全な活用
- 各種 eBPF プログラムの Kubernetes を用いた管理

👍 “`--privileged`” なしで eBPF プログラムをロード可能 → ある程度安全に運用できる

👍 Kubernetes DaemonSet 等を用いることでノード運用を効率化可能

課題: コンテナランタイム, k8s としては **BPF Token を扱う仕組みを持たない**

→ **自前で BPF Token に権限を付与する仕組み**が必要

※ LXD では既にサポート済み (“`security.delegate_bpf`”)

c.f. https://documentation.ubuntu.com/lxd/latest/explanation/bpf/

課題: eBPF 自体の安全性

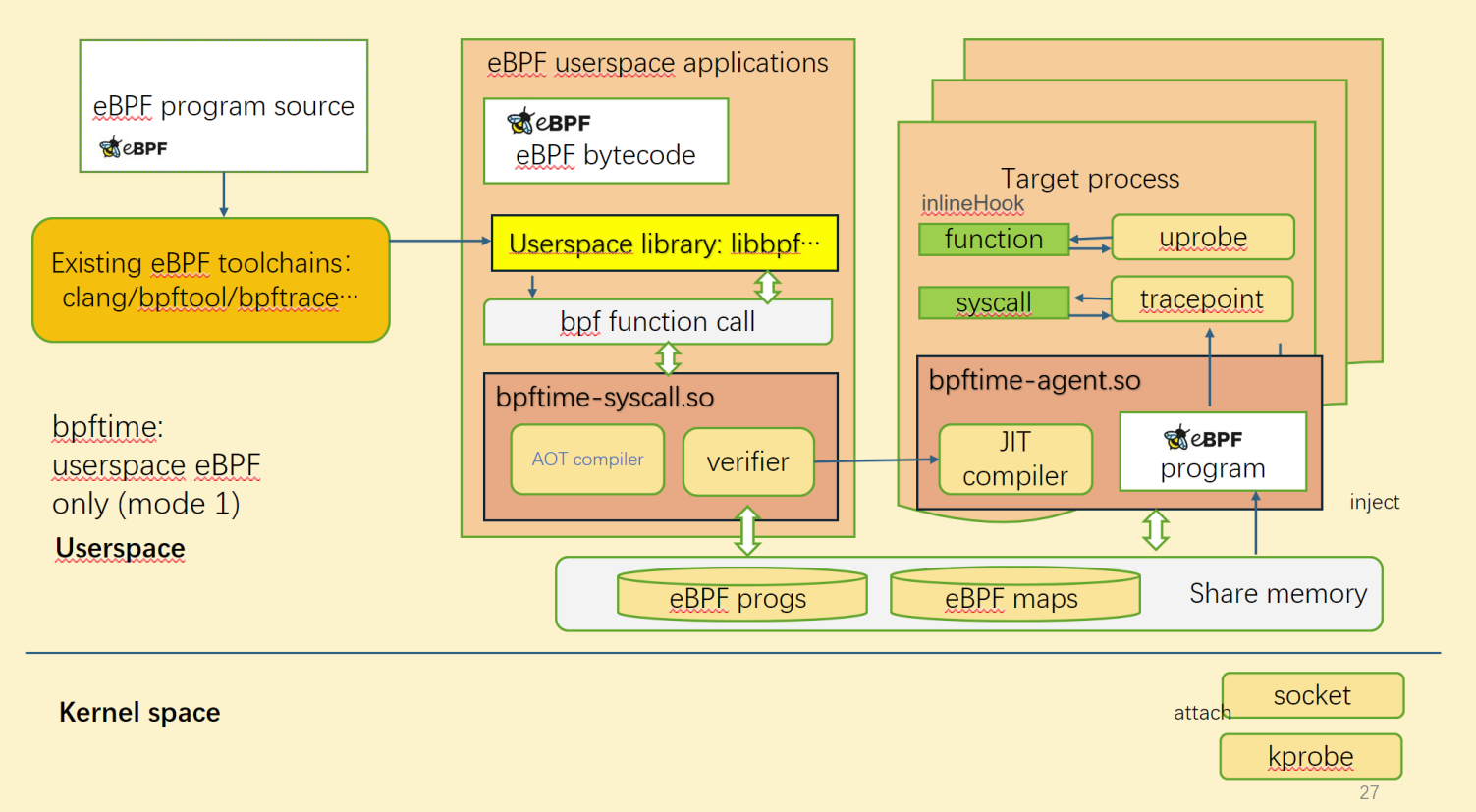
カーネル空間で動く以上、絶対に安全とは言い切れない

スライド: https://pibvt.net/s/zUEhT3GJrUc

その他

bpftime: ユーザー空間で eBPF を動かす取り組み

https://github.com/eunomia-bpf/bpftime



スライド: https://pibvt.net/s/zUEhT3GJrUc

まとめ

eBPF の利用に必要な権限を整理

- `unprivileged_bpf_disabled = 1,2` な場合
  - ソケット: `CAP_BPF`
  - ネットワーク系: `CAP_BPF+CAP_NET_ADMIN`
  - トレーシング系: `CAP_BPF+CAP_PERFMON` + α
- Verifier の挙動は付与する capability によって変化する
  - `CAP_NET_ADMIN`: 厳密なチェック(ポインタについて減算ができない等)
  - `CAP_PERFMON`: ある程度緩和されたチェック
- BPF Token を使うと Rootless コンテナ内でも eBPF を使える
  - ホスト側で BPF Token に対して権限を付与
  - eBPF の Map や Program の種類に応じて権限付与可能