Towards truly portable eBPF
Itay Shakury & Rafael D. Tinoco
OSS @ Aqua Security

Linux Plumbers 2021
Hello

- Tracee – runtime security using eBPF
- Tell our story of building and shipping eBPF application
- Our POV: vendor not user, targeting common users
- Go -> eBPF
- User experience > developer productivity

$ docker run aquasec/tracee
Option 1 – all in one image

Challenges:
- Long startup time
- Big image (~155MB)
- Obtain correct headers
- Fragile header discovery
Challenges:

• Adds friction to installation
• Still need to compile BPF, headers
• Deliver artifact to containers
• Management in heterogenous fleet
CO-RE

- **CONFIG_DEBUG_INFO_BTF=y**
- `/sys/kernel/btf/vmlinux`

**Libraries**
- **CLI binary**
- **Portable BPF object**
- **libbpf**

**Challenges**:
- Portability aches
- `libbpf` in Go
- BTF prevalence

> https://nakryiko.com/posts/bpf-portability-and-co-re/
func main() {
    eventsChannel := make(chan []byte, 0)
    sig := make(chan os.Signal, 1)
    signal.Notify(sig, os.Interrupt)

    // initialize
    m, err := libbpfgo.NewModuleFromFile("myprobe.bpf.o")
    must(err)
    m.BPFLoadObject()
    prog, err := m.GetProgram("execve_handler")
    must(err)
    _, err = prog.AttachTracepoint("syscalls:sys_enter_execve")
    must(err)
    rb, _ := m.InitRingBuf("events", eventsChannel)

    // start
    fmt.Println("starting")
    rb.Start()
    go func() {
        for e := range eventsChannel {
            x, err := binary.ReadUvarint(bytes.NewReader(e))
            must(err)
            fmt.Printf("event: %v\n", x)
        }
    }()

    // wait
    <-sig
    fmt.Println("stopping")
    rb.Stop()
    m.Close()
}
BTF in the wild

BPF CO-RE (Compile Once – Run Everywhere)

Libbpf supports building BPF CO-RE-enabled applications, which, in contrast to BCC, do not require Clang/LLVM runtime being deployed to target servers and doesn't rely on kernel-devel headers being available.

It does rely on kernel to be built with BTF type information, though. Some major Linux distributions come with kernel BTF already built in:

- Fedora 31+
- RHEL 8.2+
- OpenSUSE Tumbleweed (in the next release, as of 2020-06-04)
- Arch Linux (from kernel 5.7.1.arch1-1)
- Manjaro (from kernel 5.4 if compiled after 2021-06-18)
- Ubuntu 20.10
- Debian 11 (amd64/arm64)

If your kernel doesn't come with BTF built-in, you'll need to build custom kernel. You'll need:

- pahole 1.16+ tool (part of dwarves package), which performs DWARF to BTF conversion;
- kernel built with CONFIG_DEBUG_INFO_BTF=y option;
- you can check if your kernel has BTF built-in by looking for /sys/kernel/btf/vmlinux file:

```bash
$ ls -la /sys/kernel/btf/vmlinux
-rw-r--r-- 1 root root 3541561 Jun  2 18:16 /sys/kernel/btf/vmlinux
```
CONFIG_DEBUG_INFO_BTF=y

/sys/kernel/btf/vmlinux

External BTF

BTF

CO-RE

CLI binary

libbpf

BPF object
How to load external BTF?

```c
struct btf *btf_load_vmlinux_btf(void) {
    struct {
        const char *path_fmt;
        bool raw_btf;
    } locations[] = {
        /* try canonical vmlinux BTF through sysfs first */
        { "/sys/kernel/btf/vmlinux", /* true */ raw BTF */, 
        /* fall back to trying to find vmlinux ELF on disk otherwise */
        { "/boot/vmlinux-x86$", 
        { "/lib/modules/x86$/build/vmlinux", 
        { "/lib/modules/x86$/kernel/vmlinux", 
        { "/lib/modules/x86$/debug/boot/vmlinux-x86$", 
        { "/usr/lib/debug/boot/vmlinux-x86$", 
        { "/usr/lib/debug/boot/vmlinux-x86$.debug", 
        { "/usr/lib/debug/lib/modules/x86$/vmlinux", 
        {
    
    commit: 1373ff599556
    Author: Shuyi Cheng <chengshuyi@linux.alibaba.com>
    Date: Tue Jul 13 09:42:37 2021

    libbpf: Introduce 'btf_custom_path' to 'bpf_obj_open_opts'

    struct bpf_object_open_opts {
        ...
        const char *btf_custom_path;
        
        #define bpf_object_open_opts__last_field btf_custom_path
    };
```
How to generate a BTF?

```sh
$ pahole --help
Usage: pahole [OPTION...] FILE
-a, --anon_include           include anonymous classes
-A, --nested_anon_include    include nested (inside other structs) anonymous classes
--btf_base=PATH              Path to the base BTF file
--btf_encode_force           Ignore those symbols found invalid when encoding BTF.
--btf_gen_all                Allow using all the BTF features supported by pahole.
--btf_gen_floats             Allow producing BTF KIND_FLOAT entries.
-B, --bit_holes=NR_HOLES     Show only structs at least NR_HOLES bit holes
-c, --cacheline_size=SIZE    set cacheline size to SIZE
--classes_as_struct          Use 'struct' when printing classes
--count=COUNT                Print only COUNT input records.
--class_name=CLASS_NAME      Show just this class
-d, --recursive              recursive mode, affects several other flags
--decl_exclInputModule      exclude classes declared in files with PREFIX
-E, --extra_types            expand class members
-f, --find_pointers_to=CLASS_NAME
Find pointers to CLASS_NAME
--first_obj_only             Only process the first object file in the binary
--fixup_silly_bitfields      Fix silly bitfields such as int foo:32
--flat_arrays                Flat arrays
-F, --format_path=FORMAT_LIST List of debugging formats to try
--header_type=TYPE           File header type
--hex                        Print offsets and sizes in hexadecimal
--holes=NR_HOLES             show only structs with at least NR_HOLES holes
--i, --contains=CLASS_NAME   Show classes that contains CLASS_NAME
--i, --show_decl_info        Show the file and line number where the tags were defined
--j, --btf_encode_detached=FILENAME
Encode as BTF in a detached file
--kabi_prefix               Encode as BTF when the prefix of the string is STRING, treat the string as STRING.
--l, --show_first_biggest_size_base_type_member
--l, --nr_methods
--l, --show_only_data_members show first biggest size base_type member
show number of methods
show only the members that use space in the
```
BTF Generation Script - Ubuntu

# extract vmlinux file from ddeb package

dpkg --sys-tarfile "${version}.ddeb" | \
tar xvf - "../usr/lib/debug/boot/vmlinux-${version}" \ |
{       warn "could not deal with ${version}, cleaning and moving on..." \  rm -rf "${basedir}/ubuntu/${ubuntuver}/x86_64/usr" \  rm -rf "${version}.ddeb" \  touch "${version}.failed" \  continue \}

mv "../usr/lib/debug/boot/vmlinux-${version}" "./${version}.vmlinux" \ |
{       warn "could not rename vmlinux ${version}, cleaning and moving on..." \  rm -rf "${basedir}/ubuntu/${ubuntuver}/x86_64/usr" \  rm -rf "${version}.ddeb" \  touch "${version}.failed" \  continue \}

rm -rf "${basedir}/ubuntu/${ubuntuver}/x86_64/usr"

pahole -j "${version}.btf" "${version}.vmlinux"
# pahole "./${version}.btf" > "${version}.txt"
tar cvfJ "./${version}.btf.tar.xz" "${version}.btf"
CONFIG_DEBUG_INFO_BTF=y

/sys/kernel/btf/vmlinux

BTF

CLI binary

libbpf

BPF object

External BTF

pahole

download

BTF x.y.z

BTF x.y.z

https://github.com/aquasecurity/btfhub
## CentOS

<table>
<thead>
<tr>
<th>CentOS 7</th>
<th>RHEL</th>
<th>Release Date</th>
<th>RHEL Date</th>
<th>Kernel</th>
<th>BPF</th>
<th>BTF</th>
<th>HUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.0.1406</td>
<td>7.0</td>
<td>2014-07</td>
<td>2014-06-09</td>
<td>3.10.0-123</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.1.1503</td>
<td>7.1</td>
<td>2015-03</td>
<td>2015-03-05</td>
<td>3.10.0-229</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.2.1511</td>
<td>7.2</td>
<td>2015-11</td>
<td>2015-11-19</td>
<td>3.10.0-327</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.3.1611</td>
<td>7.3</td>
<td>2016-11</td>
<td>2016-11-03</td>
<td>3.10.0-514</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.4.1708</td>
<td>7.4</td>
<td>2017-07</td>
<td>2017-07-31</td>
<td>3.10.0-693</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.5.1804</td>
<td>7.5</td>
<td>2018-04</td>
<td>2018-04-10</td>
<td>3.10.0-862</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.6.1810</td>
<td>7.6</td>
<td>2018-10</td>
<td>2018-10-30</td>
<td>3.10.0-957</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.7.1908</td>
<td>7.7</td>
<td>2019-08</td>
<td>2019-08-06</td>
<td>3.10.0-1062</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.8.2003</td>
<td>7.8</td>
<td>2020-03</td>
<td>2020-03-31</td>
<td>3.10.0-1127</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>7.9.2009</td>
<td>7.9</td>
<td>2020-09</td>
<td>2020-09-29</td>
<td>3.10.0-1160</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

*Note: Latest centos7 kernels support BPF, and might support BTF, but they lack some eBPF features. With that, eBPF programs capable of running in those systems are very limited.*

Check out eBPF features your code use [HERE](#).

## Fedora

<table>
<thead>
<tr>
<th>Fedora</th>
<th>Release Date</th>
<th>Kernel</th>
<th>BPF</th>
<th>BTF</th>
<th>HUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>2018-10-30</td>
<td>4.18</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2019-05-07</td>
<td>5.0</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>2019-10-29</td>
<td>5.3</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2020-04-28</td>
<td>5.6</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>2020-10-27</td>
<td>5.8</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>2021-04-27</td>
<td>5.11</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

*Note: All supported future Fedora releases will have BPF & BTF support enabled.*

## Ubuntu

<table>
<thead>
<tr>
<th>Ubuntu Ver</th>
<th>Num</th>
<th>Release Date</th>
<th>Kernel</th>
<th>BPF</th>
<th>BTF</th>
<th>HUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bionic</td>
<td>18.04.2</td>
<td>2018-04-26</td>
<td>4.15.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bionic HWE</td>
<td>-</td>
<td>-</td>
<td>5.4.0</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>Focal</td>
<td>20.04.2</td>
<td>2020-04-23</td>
<td>5.4.0</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>Focal HWE</td>
<td>-</td>
<td>-</td>
<td>5.8.0</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>Groovy</td>
<td>20.10</td>
<td>2020-10-22</td>
<td>5.8.0</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>Groovy HWE</td>
<td>20.10</td>
<td>-</td>
<td>5.11.0</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>Hirsute</td>
<td>21.04</td>
<td>2021-04-22</td>
<td>5.11.0</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
</tr>
</tbody>
</table>

*Notes: Bionic HWE, Focal and Focal HWE kernels need this HUB. All other future Ubuntu releases will have BPF & BTF support enabled.*

### Disclaimer

BPF & BTF support enabled.
BTF Hub is open, feel free to engage

This PR is an experiment that uses https://github.com/kinvolk/inspektor-gadget to run CO-RE based tools in systems without CONFIG_DEBUG_INFO_BITFy

**How does it work?**

The entrypoint script tries to download the BTF file for the current kernel, if it’s successful it creates an ELF file with a .BTF section containing the BTF debug info and stores it at /boot/vmlinux-$uname -r

**Testing done**

I created a test cluster in ubuntu focal (using kubeadm) and tried to run some of the tools.

```
$ cat /boot/config-$uname -r | grep BTF
CONFIG_VIDEO_SONY_BTF_MPKn
# CONFIG_DEBUGINFO_BTF is not set

$ kubectl -n hub-system ls -p | grep BTFNAME
03 detected: "Ubuntu 20.04.1 LTS"
Kernel detected: 5.4.0-80-generic
Bin detected: 0.21.0-1
Gadget image: docker.io/kinvolk/gadget:mauricio-btf-hub-poc
Deployment options:
INSPEKTOR_GADGET_OPTION_TRACELOOP_LEG_LEVEL=info, json
INSPEKTOR_GADGET_OPTION_TRACELOOP=false
INSPEKTOR_GADGET_OPTION_TOOLS_MODE=auto
INSPEKTOR_GADGET_OPTION_HOOK_MODE=auto
Inspector Gadget version: v0.1.115-80+g16433-dirty
Falling back to podinfo for heck.
BTF is not available; Trying btfhub
Trying to download unilinux from https://github.com/aquasecurity/btfhub/raw/main/ubuntu/20.04/5.4.0-80-generic-unilinux downloaded. Using CO-RE based tools
Starting the Gadget Tracer Manager in the background...
```
CO-RE: Challenges

- **PORTABILITY**
  - Kernel memory access
  - Diff stack sizes
  - Loop unrolling & complexity
  - Tail calls

- **LIBBPF SUPPORT**
  - Destroy vs Detach
  - Missing legacy kprobes support
  - Destroy/detach changes

- **BTF RELOCATIONS**
  - Quick Overview
    - BPF ELF Section Headers
    - BPF ELF Symbols Table
    - Kconfig file dependency
    - Kconfig relocations
CHALLENGE: PORTABILITY
(CO-RE and different kernel versions)
CO-RE: Challenges (portability: kernel memory access)

1. **LIBBPF NON-CO-RE**
   - `bpf_probe_read(&pid, sizeof(pid), &task->pid);`

2. **LIBBPF NON-CO-RE + BPF_PROG_TYPE_TRACING (v5.4-rc3)**
   - `pid_t pid = task->pid;`

3. **LIBBPF CO-RE** (same as `bpf_probe_read()` with `__builtin_preserve_access_index()`)
   - `bpf_core_read(&pid, sizeof(pid), &task->pid);`

4. **LIBBPF CO-RE + BPF_PROG_TYPE_TRACING**
   - `__builtin_preserve_access_index()` LLVM built-in support: Accesses to aggregate data structures (structs, unions, arrays) in the argument will have appropriate CO-RE relocation information generated.
   - `pid_t pid = __builtin_preserve_access_index({task->pid});`
CO-RE: Challenges (portability: unrolling & complexity)

For the kprobe `security_sb_mount`, the `save_path_to_str_buf()` complexity is too big with the unroll logic + MAX_PATH_COMPONENTS of 80, even on higher kernels (like 5.4 in Ubuntu). Reducing to 64 did NOT help. Reducing to 48 DID help and it worked.

I checked Ubuntu kernel and it contains c04cd2d968a ("bpf: increase complexity limit and maximum program size") commit with no reversions, which indicates that we might need to either split that logic into tails OR define less than 80 for 5.4.x kernels (if others are good with that number).

Based on the commit:

```
#define BPF_COMPLEXITY_LIMIT_INSNS 48
```

it might be that the eBPF verifier in older kernels, like 5.4, is not smart enough to consider an unroll of 80 iterations, in the path resolution function, a logic less complex than it should.

So, we can do:

```
#define MAX_PATH_COMPONENTS 48
```

or change the defaults. Up to you! This small change fixes the issue for NON CO-RE runs in the Ubuntu 5.4 kernel.

```
static __always_inline int save_path_to_str_buf(buf_t *string_p, const struct path *path) {
    struct path f_path;
    bpf_probe_read(&f_path, sizeof(struct path), path);
    char slash = '/';
    int zero = 0;
    struct dentry *dentry = f_path.dentry;
    struct vfsmnt *vfsmnt = f_path.vmnt;
    struct mount *mnt_parent_p;
    struct mount *mnt_p = real_mount(vfsmnt);
    bpf_probe_read(&mnt_p->mnt_parent_p, sizeof(struct mount*), &mnt_p->mnt_parent);

    u32 buf_off = (MAX_PERCPU_BUFSIZE >> 1);
    struct dentry *mnt_root;
    struct dentry *d_parent;
    struct qstr d_name;
    unsigned int len;
    unsigned int off;
    int sz;

    /*pragma unroll*/
    for (int i = 0; i < MAX_PATH_COMPONENTS; i++) {
        mnt_root = get_mnt_root_ptr_from_vfsmnt(vfsmnt);
        d_parent = get_d_parent_ptr_from_dentry(dentry);
        if (dentry == mnt_root || dentry == d_parent) {
            if (dentry != mnt_root) {
                // We reached root, but not mount root - escaped?
                break;
            }
            if (mnt_p != mnt_parent_p) {
                // We reached root, but not global root - continue with mount point path
                bpf_probe_read(dentry, sizeof(struct dentry*), d.entry->mntpoint);
                bpf_probe_read(*mnt_p->mntpoint);   
                bpf_probe_read(&mnt_parent_p, sizeof(struct mount*), &mnt_p->mnt_parent);
                vfsmnt = &mnt_p->vmsnt;
                continue;
            }
            // Global root - path fully parsed
            break;
        }
        // Add this dentry name to path
        d_name = get_d_name_from_dentry(dentry);

        switch (d_name.type) {
```
CO-RE: Challenges (portability: tail calls complexity)

```c
for e := range t.eventsToTrace {
  eUS2 := uint32(e) // e is int32
  params := eventsParams[e]
  paramTypes := uint64
  var paramNames uint64
  for n, param := range params {
    paramTypes = paramTypes | (uint64(param.encType) << (8 * n))
    paramNames = paramNames | (uint64(param.encName) << (8 * n))
  } // if err := paramsTypesBPFMap.Update(unsafe.Pointer(&eUS2),
  // unsafe.Pointer(&paramTypes)); err != nil {
  // return err
  // if err := paramsNamesBPFMap.Update(unsafe.Pointer(&eUS2),
  // unsafe.Pointer(&paramNames)); err != nil {
  if e == ExecEvEventID || e == ExecEvEventID {
    event, ok := EventsIDTOEvent[e]
    if !ok {
      continue
    } // if execute functions require tail call on syscall enter as they perform extra work
    probFnName := fmt.Sprintf("syscall_%s", event.Name)
    err = t.intCall(uint32(e), "exec_event", probFnName)
    if err != nil {
      return err
    } // if err = t.intTailCall(uint32(e), "sys_exit_tails", probFnName) // if ever needed
  }

  SEC("raw_tracepoint/sys_enter")
  int tracpoint__raw_syscalls_sys_enter(struct bpf__raw_tracepoint_args *ctxx) {
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    ... // call syscall handler, if exists
    // enter tail calls should never delete saved args
    bpf_tail_call(ctxx, &sys_enter_tails, id);
    return 0;
  }
```
CHALLENGE: LIBBPF SUPPORT
(1:1 libbpfgo & libbpf)
CO-RE: Challenges (libbpf support: link destroy vs detach)

- **commit d88b71d4a916** libbpf: remove unused bpf_link’s destroy operation, add dealloc

  ```c
  /* Release "ownership" of underlying BPF resource (typically, BPF program
   * attached to some BPF hook, e.g., tracepoint, kprobe, etc). Disconnected
   * link, when destructed through bpf_link_destroy() call won't attempt to
   * detach/unregister that BPF resource. This is useful in situations where,
   * say, attached BPF program has to outline userspace program that attached it
   * in the system. Depending on type of BPF program, though, there might be
   * additional steps (like pinning BPF program in BPF FS) necessary to ensure
   * exit of userspace program doesn't trigger automatic detachment and clean up
   * inside the kernel.
   */

  void bpf_link__disconnect(struct bpf_link *link)
  {
    link->disconnected = true;
  }

  int bpf_link__destroy(struct bpf_link *link)
  {
    int err = 0;
    if (IS_ERR_OR_NULL(link))
      return 0;
    if (!link->disconnected && link->detach)
      err = link->detach(link);
    if (link->pin_path)
      free(link->pin_path);
    if (link->dealloc)
      link->dealloc(link);
    else
      free(link);
    return libbpf_err(err);
  }
  ```

  // get BPF program from BPF object
  bpfProgKsysSync, err = bpfModule.GetProgram("ksys_sync")
  if err != nil {
    errExit(err)
  }
  // attach to BPF program to kprobe
  bpfLinKsysSync, err := bpfProgKsysSync.AttachKprobe("ksys_sync")
  if err != nil {
    errExit(err)
  }
  // test detaching (libbpfgo PR #78 from Geyslan)
  err = bpfLinKsysSync.Detach()
  if err != nil {
    errExit(err)
  }

- **mine.go : 97 “invalid argument”:**
  bpf_link__link_detach() shouldn’t be used directly.

- **link->destroy() usage is tricky:**

  you may **disconnect bpf_link** and destroy internal resources only, keeping perf event fd opened and event enabled.
CO-RE: Challenges (libbpf support: legacy kprobe interface)

commit 668ace0ea5ab libbpf: use BPF perf link when supported by kernel

DIFFERENT INTERFACES FOR EBPF LINK ATTACHMENTS TO PROBES AND TRACEPOINTS:

1. PERF_EVENT_IOC_SET_BPF (attaches program to existing kprobe tracepoint event) + PERF_EVENT_IOC_ENABLE (enables event specified by fd).
2. BPF_LINK_CREATE (for-next tree)
3. LEGACY KPROBE_EVENTS (for-next tree)
**CO-RE: Challenges (libbpf support: legacy kprobe interface)**

**commit 155f556d4b1**
**Author:** Rafael David Tinoco <rafaeldtinoco@ubuntu.com>
**Date:** Tue Mar 23 01:09:52 2021

libbpf: Add bpf object kern_version attribute setter

Unfortunately some distros don't have their kernel version defined accurately in `<linux/version.h>` due to different long term support reasons.

It is important to have a way to override the bpf kern_version attribute during runtime: some old kernels might still check for kern_version attribute during bpf_prog_load().

**commit ca304b40c20d**
**Author:** Rafael David Tinoco <rafaeldtinoco@gmail.com>
**Date:** Sun Sep 12 03:48:44 2021

libbpf: Introduce legacy kprobe events support

Allow kprobe tracepoint events creation through legacy interface, as the kprobe dynamic PMUs support, used by default, was only created in v4.17.

Store legacy kprobe name in struct bpf_perf_link, instead of creating a new "subclass" off of bpf_perf_link. This is ok as it's just two new fields, which are also going to be reused for legacy uprope support in follow up patches.

**commit 46ed5fc33db9**
**Author:** Andrii Nakryiko <andrii@kernel.org>
**Date:** Tue Sep 21 18:00:35 2021

libbpf: Refactor and simplify legacy kprobe code

This patch also implicitly fixes the problem with invalid open() error handling present in poke_kprobe_events(), which (the function) this patch removes.

Fixes: ca304b40c20d ("libbpf: Introduce legacy kprobe events support")

Kernel v4.15 needs eBPF `kern_version` attribute (we’re currently supporting v4.19 and on).

Kernel v4.19 still needs kprobe points to be added to `kprobe_events` (legacy kprobe support to libbpf) – thanks Andrii for reviewing and accepting it.

Note: Last days Andrii simplified legacy kprobe code and introduced legacy uprope support, besides fixing some issues.
(quick pause: eBPF and relocations)
CO-RE: BPF Section Headers (quick overview)

Sections:
Idx  Name                        Size         VMA                                Type              Value
 0    .text                       00000000  00000000000000000 TEXT
 1    kprobe/ksys_sync            00000000  00000000000000000 TEXT
 2    tracepoint/syscalls/sys_ent  00000000  00000000000000000 TEXT
 3    tracepoint/syscalls/sys_ent  00000000  00000000000000000 TEXT
 4    license                     00000000  00000000000000000 DATA
 5    .maps                       00000000  00000000000000000 DATA
 6    .BTF                        00000000  00000000000000000 DATA
 7    .BTF .ext                   00000000  00000000000000000 DATA
 8    .symtab                     00000000  00000000000000000 DATA
 9    .relkprobe/ksys_sync         00000000  00000000000000000 DATA
10   .reltracepoint/syscalls/sys_  00000000  00000000000000000 DATA
11   .rel.BTF                     00000000  00000000000000000 DATA
12   .rel.BTF .ext                00000000  00000000000000000 DATA
13   .llvm_addrsig                00000000  00000000000000000 DATA
14   .strtab                     00000000  00000000000000000 DATA

Symbol table 'symtab' contains 12 entries:
Num: Value Size Type Bind Vis Ndx Name
 0: 00000000000000000 0 NOTYPE LOCAL DEFAULT UND
 5: 00000000000000000 0 SECTION LOCAL DEFAULT 2 kprobe/ksys_sync
 6: 00000000000000000 0 SECTION LOCAL DEFAULT 3 tracepoint/syscalls/sys_ent
 7: 00000000000000000 0 NOTYPE GLOBAL DEFAULT UND CONFIG_ARCH_HAS_SYSCALL_WRAPPER
 8: 00000000000000000 4 OBJECT GLOBAL DEFAULT 4 LICENSE
 9: 00000000000000000 24 OBJECT GLOBAL DEFAULT 5 events
10: 00000000000000000 400 FUNC GLOBAL DEFAULT 2 ksys_sync
11: 00000000000000000 400 FUNC GLOBAL DEFAULT 3 tracepoint__sys_ent

RELOCATION RECORDS FOR [kprobe/ksys_sync]:
OFFSET TYPE VALUE
 0  R_BPF_64_64 CONFIG_ARCH_HAS_SYSCALL_WRAPPER
 0  R_BPF_64_64 events

RELOCATION RECORDS FOR [tracepoint/syscalls/sys_ent]:
OFFSET TYPE VALUE
 0  R_BPF_64_64 CONFIG_ARCH_HAS_SYSCALL_WRAPPER
 0  R_BPF_64_64 events
CHALLENGE: LIBBPF SUPPORT
(kconfig relocations)
CO-RE: Challenges (BPF relocations: kconfig & dead code)

- **Kconfig relocations made with externs and eBPF map:**

  Relocation section `'.relraw_tracepoint/sys_enter' at offset 0x1624e8 contains 50 entries:

  ```
  0: 0000000000000000 0: 0000000000000000 0: 0000000000000000 0: 0000000000000000 0: 0000000000000000
  1: 0000000000000000 0: 0000000000000000 0: 0000000000000000 0: 0000000000000000 0: 0000000000000000
  ```

  Symbol table `.symbtab` contains 237 entries:

  ```
  Num: Value Type Bind Vis Name
  0: 0000000000000000 0 NOTYPE LOCAL DEFAULT
  1: 0000000000000000 0 NOTYPE LOCAL DEFAULT
  ```

  - Dead code elimination did not work for <= v5.4 kernels (constant coming from R/O map value). Verifier would not allow load because of bad accesses coming from dead branch.

```c
SEC("raw_tracepoint/sys_enter")
int tracepoint__raw_sys_enter(struct bpf_raw_tracepoint_args *ctx)
{
    int id = ctx->args[1];
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();

    if (CONFIG_ARCH_HAS_SYSCALL_WRAPPER) {
        struct pt_regs regs = {};
        bpf_probe_read(&regs, sizeof(struct pt_regs), (void*)ctx->args[0]);

        if (is_x86_compat(task)) {
            #ifdef __bpf_target_x86
            args_tmp.args[0] = regs.bx;
            args_tmp.args[1] = regs.cx;
            #endif // __bpf_target_x86
        } else {
            args_tmp.args[0] = PT_REGS_PARM1(&regs);
            args_tmp.args[1] = PT_REGS_PARM2(&regs);
            args_tmp.args[2] = PT_REGS_PARM3(&regs);
            #ifdef __bpf_target_x86
            args_tmp.args[3] = PT_REGS_PARM1(&regs);
            #endif
            ```

```
Propose the dead code verifier fix to **stable v5.4 branch**: And fixed the CO-RE issue we had. But there is more...
CO-RE: Challenges (BPF relocations: kconfig dependency)

Libbpf relocations depend on:
- **KCFG extern**
  - /proc/config.gz
  - /boot/config-$(uname -r)
- **KSYM extern** (subsequent slides)
- **RAW BTF** or ELF with .BTF sec

Libbpf allows specifying kconfig file, but it is read as *extra kconfig options*, not a ‘replacement’ for existing kconfig.gz.

**SOLUTION** was to create our own `kconfig_map`. (approach is like what libbpf does)
CHALLENGE: LIBBPF SUPPORT
(ksym relocations in any env)
Tracee with BTF Hub

[rafaeldtinoco@bionic:~].../aqasec-tracee/tracee-ebpf]$ sudo ./dist/tracee-ebpf --debug --trace uid=1000 --trace pid=new --trace event=execve

BTF: ubuntu 18.04 5.4.0-84-generic
BTF: vmlinux = false btfhub = true btfcached = false
BTF: bpfenv = false btfenv = false vmlinux = false btfhub = true btfcached = false
BTF: btfhub: https://github.com/aquasecurity/btfhub/raw/main/ubuntu/18.04/x86_64/5.4.0-84-generic.btf.tar.xz
BTF: btf file is now cached
BTF: using btf from btfhub: %s /tmp/tracee/5.4.0-84-generic.btf
BPF: using embedded bpf object
unpacked CO:RE bpf object file into memory

<table>
<thead>
<tr>
<th>TIME</th>
<th>UID</th>
<th>COMM</th>
<th>PID</th>
<th>TID</th>
<th>RET</th>
<th>EVENT</th>
<th>ARGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>05:41:39:61619</td>
<td>1000</td>
<td>bash</td>
<td>4011</td>
<td>4011</td>
<td>0</td>
<td>execve</td>
<td>pathname: /bin/cat, argv: [cat /proc/cmdline]</td>
</tr>
<tr>
<td>05:41:41:968996</td>
<td>1000</td>
<td>bash</td>
<td>4015</td>
<td>4015</td>
<td>0</td>
<td>execve</td>
<td>pathname: /sbin/ip, argv: [ip addr list]</td>
</tr>
<tr>
<td>05:41:44:197013</td>
<td>1000</td>
<td>bash</td>
<td>4017</td>
<td>4017</td>
<td>0</td>
<td>execve</td>
<td>pathname: /bin/ps, argv: [ps]</td>
</tr>
<tr>
<td>05:41:45:113015</td>
<td>1000</td>
<td>bash</td>
<td>4019</td>
<td>4019</td>
<td>0</td>
<td>execve</td>
<td>pathname: /bin/ls, argv: [ls --color=auto]</td>
</tr>
<tr>
<td>05:41:48:175616</td>
<td>1000</td>
<td>bash</td>
<td>4023</td>
<td>4023</td>
<td>0</td>
<td>execve</td>
<td>pathname: /bin/cat, argv: [cat /proc/cmdline]</td>
</tr>
<tr>
<td>05:41:53:936277</td>
<td>1000</td>
<td>bash</td>
<td>4025</td>
<td>4025</td>
<td>0</td>
<td>execve</td>
<td>pathname: /bin/true, argv: [/bin/true]</td>
</tr>
</tbody>
</table>

End of events stream
Stats: {eventCounter:6 errorCounter:0 lostEvCounter:0 lostWrCounter:0 lostNtCounter:0}

[rafaeldtinoco@bionic:~].../aqasec-tracee/tracee-ebpf]$ sudo ./dist/tracee-ebpf --debug --trace uid=1000 --trace pid=new --trace event=execve

BTF: ubuntu 18.04 5.4.0-84-generic
BTF: vmlinux = false btfhub = true btfcached = true
BTF: bpfenv = false btfenv = false vmlinux = false btfhub = true btfcached = true
BTF: using btf from btfhub: %s /tmp/tracee/5.4.0-84-generic.btf
BPF: using embedded bpf object
unpacked CO:RE bpf object file into memory

<table>
<thead>
<tr>
<th>TIME</th>
<th>UID</th>
<th>COMM</th>
<th>PID</th>
<th>TID</th>
<th>RET</th>
<th>EVENT</th>
<th>ARGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>05:42:05:436923</td>
<td>1000</td>
<td>bash</td>
<td>4044</td>
<td>4044</td>
<td>0</td>
<td>execve</td>
<td>pathname: /bin/ls, argv: [ls --color=auto]</td>
</tr>
</tbody>
</table>

End of events stream
Stats: {eventCounter:1 errorCounter:0 lostEvCounter:0 lostWrCounter:0 lostNtCounter:0}
Build other eBPF programs #100

I am interested in using the `driverkit` to build other eBPF programs besides Falco (currently working with `Tracee`).

I would like to propose some changes to make `driverkit` more generic, and I can implement those if the maintainers agree:

- Add custom templates for the eBPF builders
- Use different docker images for the eBPF builders
- Add options to disable some actions for the `driverkit` docker process (for prepare driver config and makefile templates)

And the main change: implement a way to use linux headers for all the kernel releases.

Now `driverkit` is looking for distros from ubuntu repositories (https://mirrors.edge.kernel.org/ubuntu/pool/main/linux) with next versions of kernels: ... 4.4.x, 4.15.x, 5.4.x, 5.8.x, 5.11.x.

We need to support all the kernel versions.

Happy to hear what you think and if you have other suggestions for supporting this use case!
Fix the artifact for the environment (cross-compilation)

Fix the environment for the artifact (external BTF)
Truly portable eBPF application

✅ Smaller BTFs

Less BTFs
CONFIG_DEBUG_INFO_BTF=y

/sys/kernel/btf/vmlinux

BTF

CLI binary
libbpf
BPF object

External BTF

My BTF x.y.1-12
My BTF x.y.12-16

debuginfo x.y.z
pahole

BTF x.y.z
BTF x.y.z
Let us know what you think

- https://github.com/aquasecurity/tracee
- https://github.com/aquasecurity/libbpfgo
- https://github.com/aquasecurity/btfhub
- Itay Shakury @itaysk
- Rafael D. Tinoco @rafaeldtinoco
- Yaniv Agman @AgmanYaniv
- Grant Seltzer @GrantSeltzer
References

https://nakryiko.com/posts/bpf-portability-and-co-re/
https://nakryiko.com/posts/btf-dedup/
https://lwn.net/Articles/801479/
https://github.com/libbpf/libbpf#bpf-co-re-compile-once--run-everywhere
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications

Images and Icons:
https://commons.wikimedia.org/
https://www.flaticon.com/