

LINUX SECURITY SUMMIT EUROPE

vock

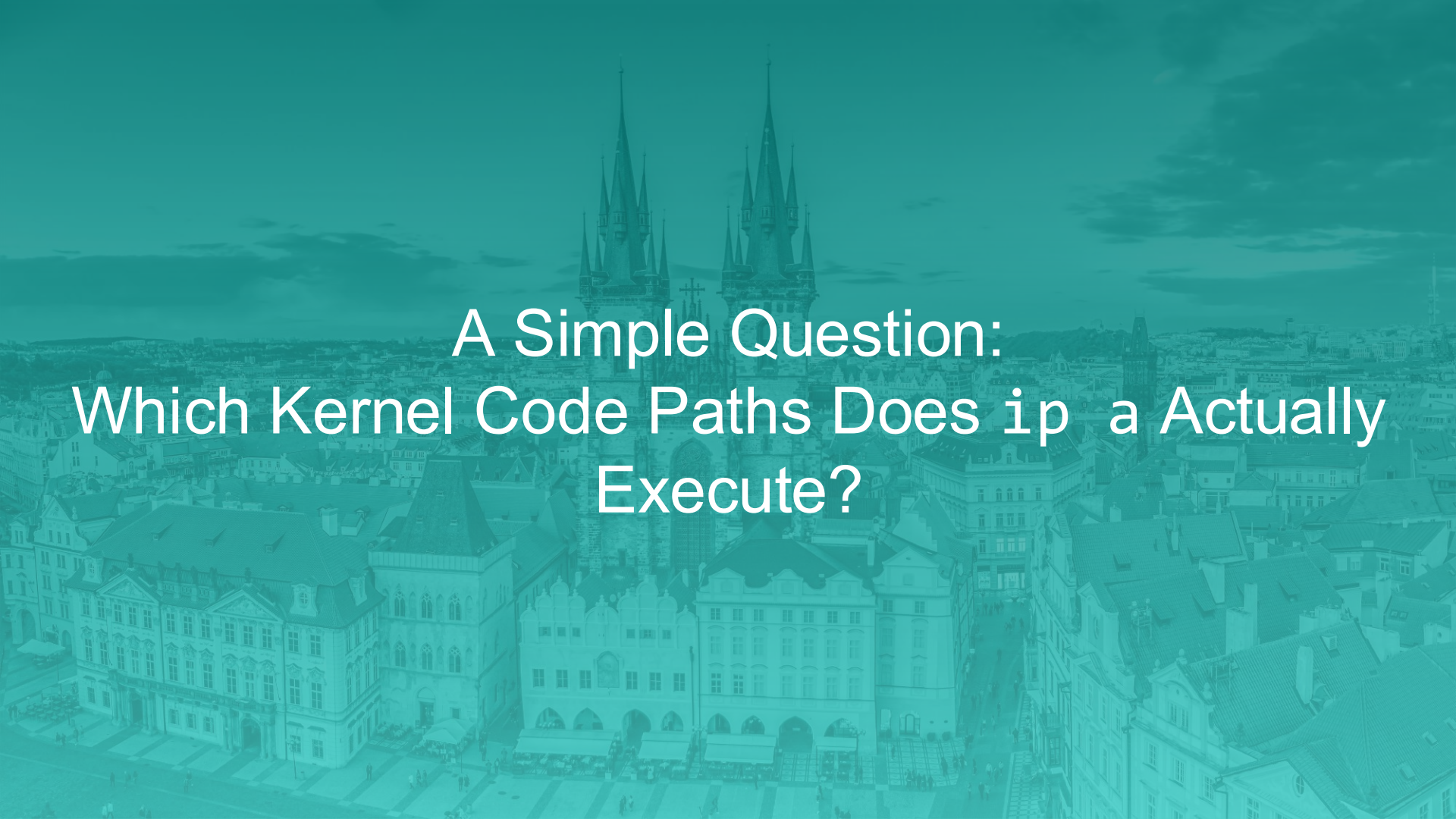
Mapping Kernel Attack Surface With Zero-Configuration Coverage

Yunseong Kim <yunseong.kim@est.tech>

Index

1. A Simple Question: Which Kernel Code Block does `/bin/ip a` Actually Execute?
2. Kernel Coverage Collection on a KCOV-Enabled Kernel
3. Motivation: Collecting Kernel Coverage for Forked and Multi-Threaded Target Programs
4. Motivation: How Can We Collect Kernel Coverage Without Enabling KCOV?

- Additional Features for Security Research: Built-in DWARF reader, syzlang Interpretation and more



A Simple Question: Which Kernel Code Paths Does `ip a` Actually Execute?

A Simple Question: Which Kernel Code Paths Does ip a Actually Execute?

- strace
  - Traces only the syscall userspace/kernel boundary.
- ftrace
  - Lacks per-hook context gating (polluted by interrupts) and operates at coarse function granularity rather than fine-grained basic block paths.
- perf
  - Can drive hardware tracers (Intel PT, AMD IBS, Arm CoreSight), but fundamentally relies on a profiler's weighted sampling model.
- JTAG Debuggers
  - Requires physical hardware. Sometime needs enterprise licenses.

Getting Started with zero configuration

```
# vock --mode hw \
  --kernel-src ~/linux-source \
  --vmlinux /path/to/vmlinux \
    ip a
```

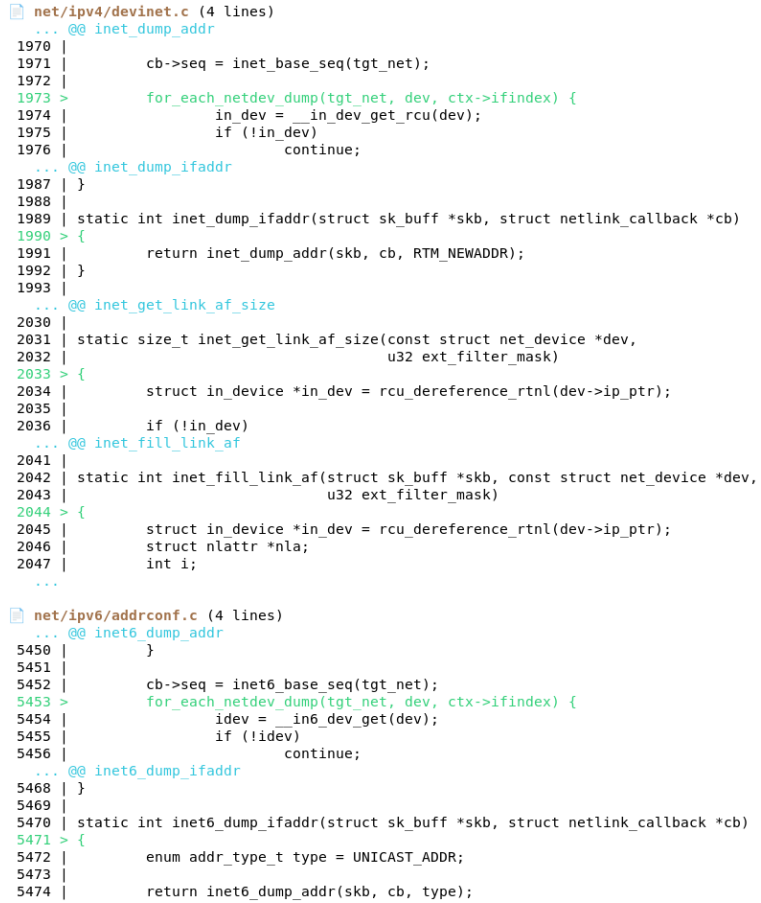
- `--kernel-src` maps DWARF paths to source files
- `--vmlinux` provides symbols and debug information
- `--mode hw` selects CPU vendor-appropriate backend
  - Coverage engine. hw is the default.

```
$ sudo vock --mode hw --kernel-src  ~/linux-7.1.3/ --vmlinux /usr/lib/debug/lib/modules/7.1.3+deb13-amd64/vmlinux ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: wlp0s20f3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 80:e4:ba:46:c5:a8 brd ff:ff:ff:ff:ff:ff
    altname wlx80e4ba46c5a8
    inet 172.20.10.46/20 brd 172.20.15.255 scope global dynamic noprefixroute wlp0s20f3
       valid_lft 84667sec preferred_lft 84667sec
    inet6 fe80::7b77:d03b:38ec:bf89/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: gpd0: <POINTOPOINT,MULTICAST,NOARP> mtu 1400 qdisc fq_codel state DOWN group default qlen 500
    link/none
6: ve-debian-sid@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 6e:b5:d0:d3:d5:68 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.254.138/16 metric 2048 brd 169.254.255.255 scope link ve-debian-sid
       valid_lft forever preferred_lft forever
    inet 192.168.98.129/28 brd 192.168.98.143 scope global ve-debian-sid
       valid_lft forever preferred_lft forever
    inet6 fe80::6cb5:d0ff:fed3:d568/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
[vock] intel_pt: 5356 kernel PCs → kerncov.log
============================ VOCK Coverage Reporter ============================

📊 [VOCK] Generating coverage report...

  KASLR detected: offset 0x39a00000
  5356 PCs, 821 unique symbolized, in-process DWARF (0.3s)
  186 files, 650 source lines (0.0s)

------------------------------ Coverage Report -------------------------------
```



net/ipv4/devinet.c (4 lines)
... @@ inet_dump_addr
1970 |
1971 | cb->seq = inet_base_seq(tgt_net);
1972 |
1973 >   for_each_netdev_dump(tgt_net, dev, ctx->ifindex) {
1974 | in_dev = __in_dev_get_rcu(dev);
1975 | if (!in_dev)
1976 | continue;
... @@ inet_dump_ifaddr
1987 | }
1988 |
1989 | static int inet_dump_ifaddr(struct sk_buff *skb, struct netlink_callback *cb)
1990 > {
1991 | return inet_dump_addr(skb, cb, RTM_NEWADDR);
1992 | }
1993 |
... @@ inet_get_link_af_size
2030 |
2031 | static size_t inet_get_link_af_size(const struct net_device *dev,
2032 | u32 ext_filter_mask)
2033 > {
2034 | struct in_device *in_dev = rcu_dereference_rtnl(dev->ip_ptr);
2035 |
2036 | if (!in_dev)
... @@ inet_fill_link_af
2041 |
2042 | static int inet_fill_link_af(struct sk_buff *skb, const struct net_device *dev,
2043 | u32 ext_filter_mask)
2044 > {
2045 | struct in_device *in_dev = rcu_dereference_rtnl(dev->ip_ptr);
2046 | struct nlattr *nla;
2047 | int i;
...

net/ipv6/addrconf.c (4 lines)
... @@ inet6_dump_addr
5450 | }
5451 |
5452 | cb->seq = inet6_base_seq(tgt_net);
5453 >   for_each_netdev_dump(tgt_net, dev, ctx->ifindex) {
5454 | idev = __in6_dev_get(dev);
5455 | if (!idev)
5456 | continue;
... @@ inet6_dump_ifaddr
5468 | }
5469 |
5470 | static int inet6_dump_ifaddr(struct sk_buff *skb, struct netlink_callback *cb)
5471 > {
5472 | enum addr_type_t type = UNICAST_ADDR;
5473 |
5474 | return inet6_dump_addr(skb, cb, type);
... @@ recvmsg_copy_msghdr
2882 | msg_compat = (struct compat_msghdr __user *) umsg;
2883 | err = get_compat_msghdr(msg, msg_compat, uaddr, iov);
2884 | } else {
2885 >   err = copy_msghdr_from_user(msg, umsg, uaddr, iov);
2886 | }
2887 | if (err < 0)
2888 | return err;
... @@ ___sys_recvmsg
2916 | if (unlikely(nosec))
2917 | err = sock_recvmsg_nosec(sock, msg_sys, flags);
2918 | else
2919 >   err = sock_recvmsg(sock, msg_sys, flags);
2920 |
2921 | if (err < 0)
2922 | goto out;
... @@ ___sys_recvmsg
2959 | return err;
2960 |
2961 | err = ____sys_recvmsg(sock, msg_sys, msg, uaddr, flags, nosec);
2962 >   kfree(iov);
2963 | return err;
2964 | }
2965 |
... @@ __sys_recvmsg
2991 | if (unlikely(!sock))
2992 | return -ENOTSOCK;
2993 |
2994 >   return ___sys_recvmsg(sock, msg, &msg_sys, flags, 0);
2995 | }
2996 |
2997 | SYSCALL_DEFINE3(recvmsg, int, fd, struct user_msghdr __user *, msg,
...

security/apparmor/domain.c (3 lines)
... @@ find_attach
422 | aa_state_t state;
423 | struct aa_perms *perms;
424 |
425 >   state = aa_dfa_leftmatch(attach->xmatch->dfa,
426 | attach->xmatch->start[AA_CLASS_XMATCH],
427 | name, &count);
428 | perms = aa_lookup_perms(attach->xmatch, state);
... @@ profile_transition
689 | }
690 |
691 | if (profile_unconfined(profile)) {
692 >   new = find_attach(bprm, profile->ns,
693 | &profile->ns->base.profiles, name, &info);
694 | /* info set -> something unusual that we should report
695 | * Currently this is only conflicting attachments, but other
... @@ apparmor_bprm_creds_for_exec
963 | new = handle_onexec(subj_cred, label, ctx->onexec, ctx->token,
964 | bprm, buffer, &cond, &unsafe);

```
# vock --mode hw --kernel-src ~/linux-7.1.3/ \
  --vmlinux /usr/lib/debug/lib/modules/7.1.3+deb13-amd64/vmlinux \
  ip a
```

```
security/security.c (5 lines)
  ... @@ security_bprm_creds_for_exec
 780 |  */
 781 | int security_bprm_creds_for_exec(struct linux_binprm *bprm)
 782 | {
 783 >         return call_int_hook(bprm_creds_for_exec, bprm);
 784 | }
 785 |
 786 | /**
  ... @@ security_file_permission
2393 |  */
2394 | int security_file_permission(struct file *file, int mask)
2395 | {
2396 >         return call_int_hook(file_permission, file, mask);
2397 | }
2398 |
2399 | /**
  ... @@ security_file_open
2740 |  */
2741 | int security_file_open(struct file *file)
2742 | {
2743 >         return call_int_hook(file_open, file);
2744 | }
2745 |
2746 | /**
  ... @@ security_task_prctl
3325 |         int rc = LSM_RET_DEFAULT(task_prctl);
3326 |         struct lsm_static_call *scall;
3327 |
3328 >         lsm_for_each_hook(scall, task_prctl) {
3329 |                 thisrc = scall->hl->hook.task_prctl(option, arg2, arg3, arg4, arg5);
3330 |                 if (thisrc != LSM_RET_DEFAULT(task_prctl)) {
3331 |                         rc = thisrc;
  ... @@ security_socket_getpeersec_dgram
4464 | int security_socket_getpeersec_dgram(struct socket *sock,
4465 |                                      struct sk_buff *skb, u32 *secid)
4466 | {
4467 >         return call_int_hook(socket_getpeersec_dgram, sock, skb, secid);
4468 | }
4469 | EXPORT_SYMBOL(security_socket_getpeersec_dgram);
4470 |
  ...
```

```
security/selinux/hooks.c (1 lines)
  ... @@ selinux_inode_listxattr
3522 |         const struct cred *cred = current_cred();
3523 |
3524 |         return dentry_has_perm(cred, dentry, FILE__GETATTR);
3525 > }
3526 |
3527 | static int selinux_inode_removexattr(struct mnt_idmap *idmap,
3528 |                                      struct dentry *dentry, const char *name)
  ...
 3 assembly source lines split out to asmcov.log

✓ Written: coverage.html, srccov.log, asmcov.log, kerncov.log (excerpts)
 artifacts written (0.0s)
```

```
# vock --mode hw --kernel-src ~/linux-7.1.3/ \
  --vmlinux /usr/lib/debug/lib/modules/7.1.3+deb13-amd64/vmlinux \
  ip a
```



Kernel Coverage Collection on a KCOV-Enabled Kernel

vock is an acronym (and exact anagram) of kcov

Kernel Coverage Collection on a KCOV-Enabled Kernel

- KCOV collects and exposes kernel code coverage information
  - CONFIG_KCOV=y
  - Instrument with compiler flag -fsanitize-coverage=trace-pc
- Coverage data of a running kernel is exported via the `/sys/kernel/debug/kcov` debugfs file.
- Per-task have own coverage data.
- Coverage collected PCs sequentially.
- https://docs.kernel.org/dev-tools/kcov.html

Kernel Coverage Collection on a KCOV-Enabled Kernel

Example result:

0xffffffff8243f75b
0xffffffff8243f78c
0xffffffff8243f7d7
0xffffffff82441fef
0xffffffff82442024



with `addr2line`:

0xffffffff8243f75b __netlink_create net/netlink/af_netlink.c:625
0xffffffff8243f78c __netlink_create net/netlink/af_netlink.c:631
0xffffffff8243f7d7 __netlink_create net/netlink/af_netlink.c:642
0xffffffff82441fef __netlink_insert net/netlink/af_netlink.c:507
0xffffffff82442024 __netlink_insert net/netlink/af_netlink.c:508

Kernel Coverage Collection on a KCOV-Enabled Kernel

```
0000000000000ff0 <__netlink_create>:
{
     ff0:       e8 00 00 00 00          call   ff5 <__netlink_create+0x5>
                        ff1: R_X86_64_PLT32     __fentry__-0x4
     ff5:       41 55                   push   %r13
     ff7:       49 89 d5                mov    %rdx,%r13
     ffa:       41 54                   push   %r12
     ffc:       41 89 cc                mov    %ecx,%r12d
     fff:       55                      push   %rbp
    1000:       48 89 f5                mov    %rsi,%rbp
    1003:       53                      push   %rbx
    1004:       48 89 fb                mov    %rdi,%rbx
    1007:       48 83 ec 08             sub    $0x8,%rsp
    100b:       44 89 44 24 04          mov    %r8d,0x4(%rsp)
        sock->ops = &netlink_ops;
    1010:       e8 00 00 00 00          call   1015 <__netlink_create+0x25>
                        1011: R_X86_64_PLT32    __sanitizer_cov_trace_pc-0x4
    1015:       48 c7 45 20 00 00 00    movq   $0x0,0x20(%rbp)
    101c:       00
                        1019: R_X86_64_32S      .rodata+0x60
        sk = sk_alloc(net, PF_NETLINK, GFP_KERNEL, &netlink_proto, kern);
    101d:       44 8b 44 24 04          mov    0x4(%rsp),%r8d
    1022:       48 89 df                mov    %rbx,%rdi
    1025:       48 c7 c1 00 00 00 00    mov    $0x0,%rcx
                        1028: R_X86_64_32S      .data+0xc0
    102c:       ba c0 0c 00 00          mov    $0xcc0,%edx
    1031:       be 10 00 00 00          mov    $0x10,%esi
    1036:       e8 00 00 00 00          call   103b <__netlink_create+0x4b>
                        1037: R_X86_64_PLT32    sk_alloc-0x4
        if (!sk)
    103b:       48 85 c0                test   %rax,%rax
    103e:       74 79                   je     10b9 <__netlink_create+0xc9>
    1040:       48 89 c3                mov    %rax,%rbx
        sock_init_data(sock, sk);
    1043:       e8 00 00 00 00          call   1048 <__netlink_create+0x58>
                        1044: R_X86_64_PLT32    __sanitizer_cov_trace_pc-0x4
    1048:       48 89 de                mov    %rbx,%rsi
    104b:       48 89 ef                mov    %rbp,%rdi
    104e:       e8 00 00 00 00          call   1053 <__netlink_create+0x63>
                        104f: R_X86_64_PLT32    sock_init_data-0x4
        mutex_init(&nlk->nl_cb_mutex);
    1053:       48 8d bb a0 03 00 00    lea    0x3a0(%rbx),%rdi
    105a:       48 c7 c2 00 00 00 00    mov    $0x0,%rdx
                        105d: R_X86_64_32S      .bss+0x8
    1061:       48 c7 c6 00 00 00 00    mov    $0x0,%rsi
                        1064: R_X86_64_32S      .rodata.str1.1+0x3e
    1068:       e8 00 00 00 00          call   106d <__netlink_create+0x7d>
                        1069: R_X86_64_PLT32    __mutex_init-0x4
        nlk->dump_cb_mutex = dump_cb_mutex;
    106d:       4c 89 ab c0 03 00 00    mov    %r13,0x3c0(%rbx)
        init_waitqueue_head(&nlk->wait);
    1074:       48 8d bb 00 03 00 00    lea    0x300(%rbx),%rdi
    107b:       48 c7 c2 00 00 00 00    mov    $0x0,%rdx
                        107e: R_X86_64_32S      .bss+0x8
    1082:       48 c7 c6 00 00 00 00    mov    $0x0,%rsi
                        1085: R_X86_64_32S      .rodata.str1.1+0x50
    1089:       e8 00 00 00 00          call   108e <__netlink_create+0x9e>
                        108a: R_X86_64_PLT32    __init_waitqueue_head-0x4
        sk->sk_protocol = protocol;
    108e:       66 44 89 a3 d4 01 00    mov    %r12w,0x1d4(%rbx)
    1095:       00
        sk->sk_destruct = netlink_sock_destruct;
    1096:       48 c7 83 90 02 00 00    movq   $0x0,0x290(%rbx)
    109d:       00 00 00 00
                        109d: R_X86_64_32S      .text+0x1e80
        return 0;
    10a1:       31 db                   xor    %ebx,%ebx
    10a3:       e8 00 00 00 00          call   10a8 <__netlink_create+0xb8>
                        10a4: R_X86_64_PLT32    __sanitizer_cov_trace_pc-0x4
}
    10a8:       89 d8                   mov    %ebx,%eax
    10aa:       48 83 c4 08             add    $0x8,%rsp
:
```

```
$ objdump -drS ./net/netlink/af_netlink.o > af_netlink.S
```

Kernel Coverage Collection on a KCOV-Enabled Kernel: How it works?

- Linux kernel have exported `__sanitizer_cov_trace_pc()`
- No callback function argument. On runtime, using _RET_IP_ it collect PC address using return address value.
- Compiler shall insert this function on the **basic block**

Kernel Coverage Collection on a KCOV-Enabled Kernel: How it works?

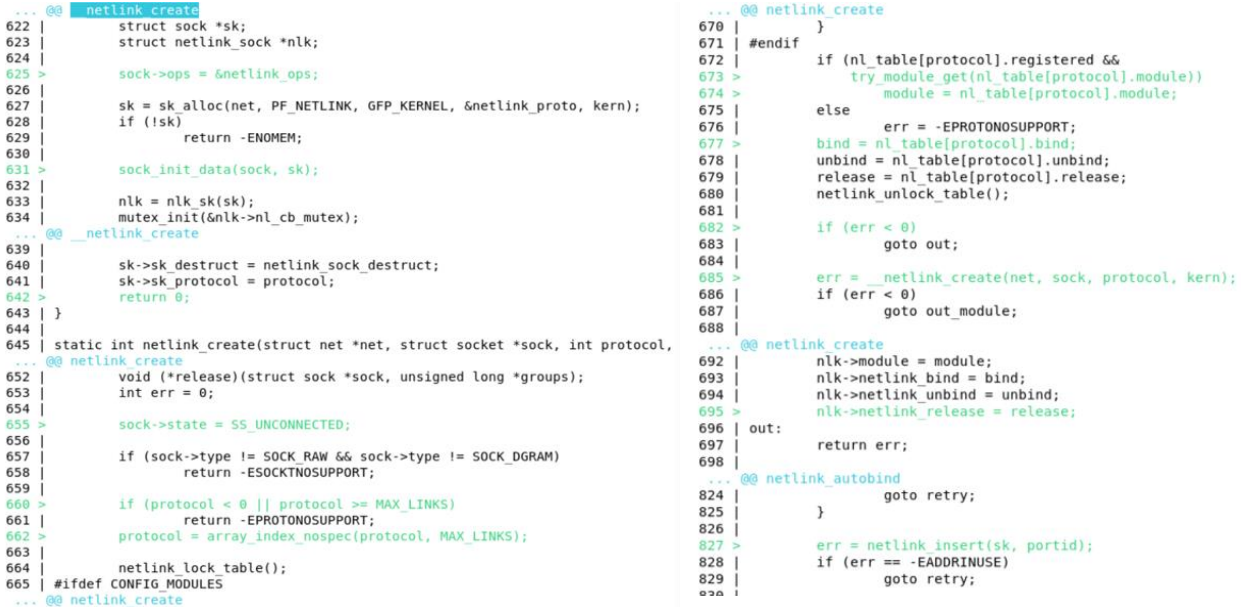
```
void foo(int *a) {
  if (a)
  *a = 0;
}
```



- It contains 3 basic blocks, let's name them A, B, C:

- Basic block is a sequence of instructions that are executed sequentially without any branches or other points of entry.

Kernel Coverage Collection on a KCOV-Enabled Kernel: vock Provides Human-Readable Visualization



```
... @@ __netlink_create
622 |          struct sock *sk;
623 |          struct netlink_sock *nlk;
624 |
625 >          sock->ops = &netlink_ops;
626 |
627 |          sk = sk_alloc(net, PF_NETLINK, GFP_KERNEL, &netlink_proto, kern);
628 |          if (!sk)
629 |                  return -ENOMEM;
630 |
631 >          sock_init_data(sock, sk);
632 |
633 |          nlk = nlk_sk(sk);
634 |          mutex_init(&nlk->nl_cb_mutex);
... @@ __netlink_create
639 |
640 |          sk->sk_destruct = netlink_sock_destruct;
641 |          sk->sk_protocol = protocol;
642 >          return 0;
643 | }
644 |
645 | static int netlink_create(struct net *net, struct socket *sock, int protocol,
... @@ netlink_create
652 |          void (*release)(struct sock *sock, unsigned long *groups);
653 |          int err = 0;
654 |
655 >          sock->state = SS_UNCONNECTED;
656 |
657 |          if (sock->type != SOCK_RAW && sock->type != SOCK_DGRAM)
658 |                  return -ESOCKTNOSUPPORT;
659 |
660 >          if (protocol < 0 || protocol >= MAX_LINKS)
661 |                  return -EPROTONOSUPPORT;
662 >          protocol = array_index_nospec(protocol, MAX_LINKS);
663 |
664 |          netlink_lock_table();
665 | #ifdef CONFIG_MODULES
... @@ netlink_create
```

```
... @@ netlink_create
670 |          }
671 | #endif
672 |          if (nl_table[protocol].registered &&
673 >              try_module_get(nl_table[protocol].module))
674 >                  module = nl_table[protocol].module;
675 |          else
676 |                  err = -EPROTONOSUPPORT;
677 >          bind = nl_table[protocol].bind;
678 |          unbind = nl_table[protocol].unbind;
679 |          release = nl_table[protocol].release;
680 |          netlink_unlock_table();
681 |
682 >          if (err < 0)
683 |                  goto out;
684 |
685 >          err = __netlink_create(net, sock, protocol, kern);
686 |          if (err < 0)
687 |                  goto out_module;
688 |
... @@ netlink_create
692 |          nlk->module = module;
693 |          nlk->netlink_bind = bind;
694 |          nlk->netlink_unbind = unbind;
695 >          nlk->netlink_release = release;
696 | out:
697 |          return err;
698 |
... @@ netlink_autobind
824 |                  goto retry;
825 |          }
826 |
827 >          err = netlink_insert(sk, portid);
828 |          if (err == -EADDRINUSE)
829 |                  goto retry;
```

```
# vock --mode kcov --vmlinux vmlinux --kernel-src linux ip a
```



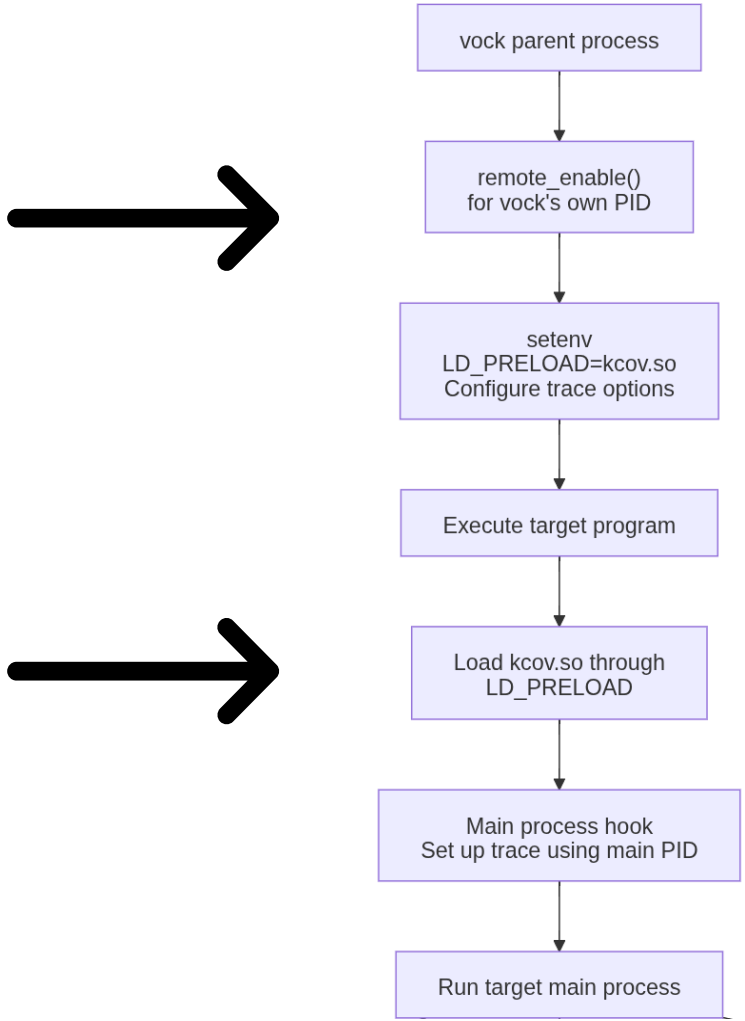
Motivation: Collecting Kernel Coverage for Forked and Multi-Threaded Target Programs

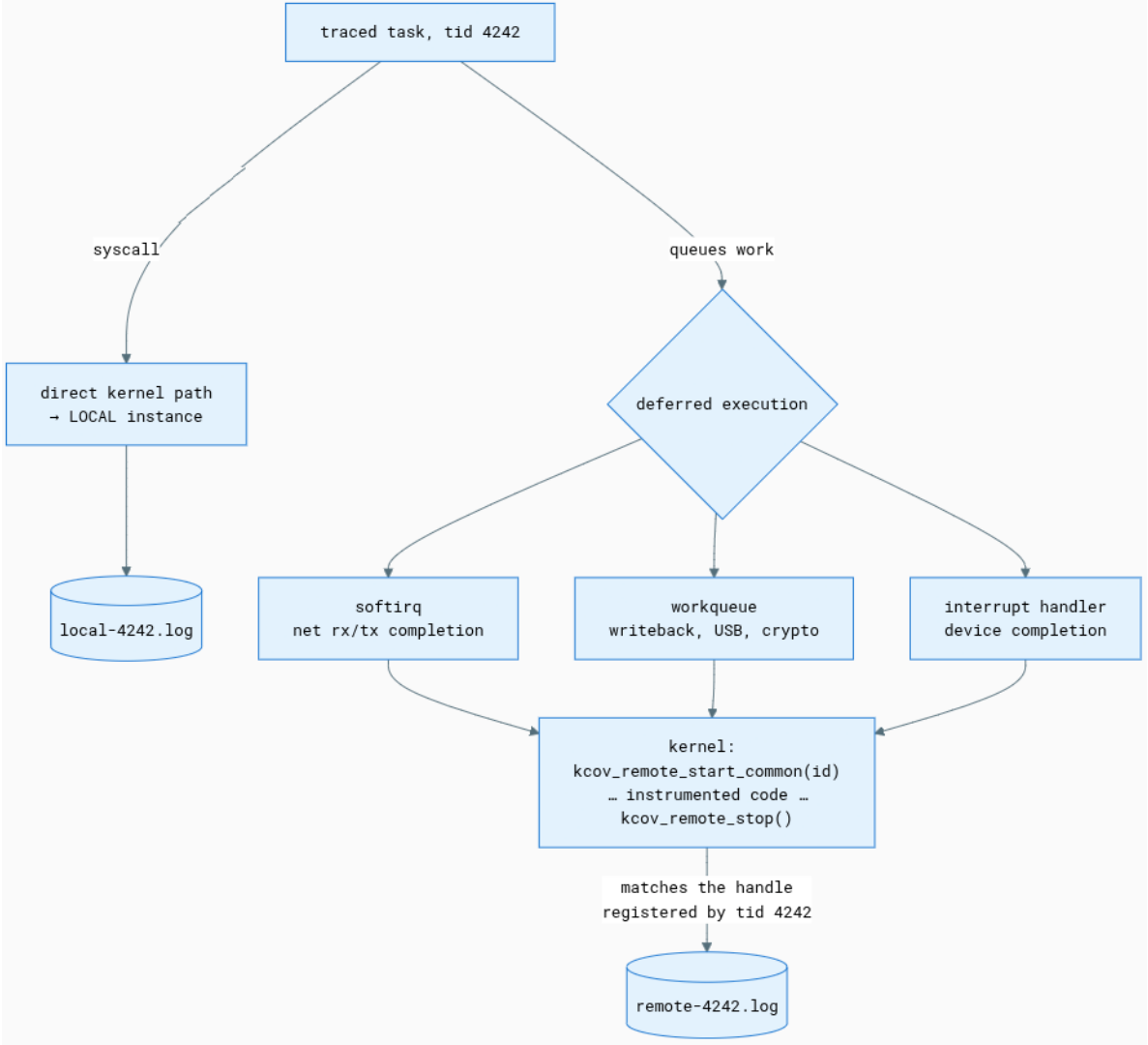
Motivation: Collecting Kernel Coverage for Forked and Multi-Threaded Target Programs

1. Remotely collected coverage is annotated with `kcov_remote_start_common()` corresponding kernel subsystems.
   - e.g., kworker, workqueue
2. Coverage can be collected separately for each forked process or thread in user space.



1. vock acts as the driving (parent) process and collects remote coverage while child processes are running.
2. LD_PRELOAD + setenv


vock parent process
remote_enable()
for vock's own PID
setenv
LD_PRELOAD=kcov.so
Configure trace options
Execute target program
Load kcov.so through
LD_PRELOAD
Main process hook
Set up trace using main PID
Run target main process


traced task, tid 4242
syscall
queues work
direct kernel path
→ LOCAL instance
deferred execution
local-4242.log
softirq
net rx/tx completion
workqueue
writeback, USB, crypto
interrupt handler
device completion
kernel:
kcov_remote_start_common(id)
… instrumented code …
kcov_remote_stop()
matches the handle
registered by tid 4242
remote-4242.log

fs/smb/server/connection.c (25 lines)

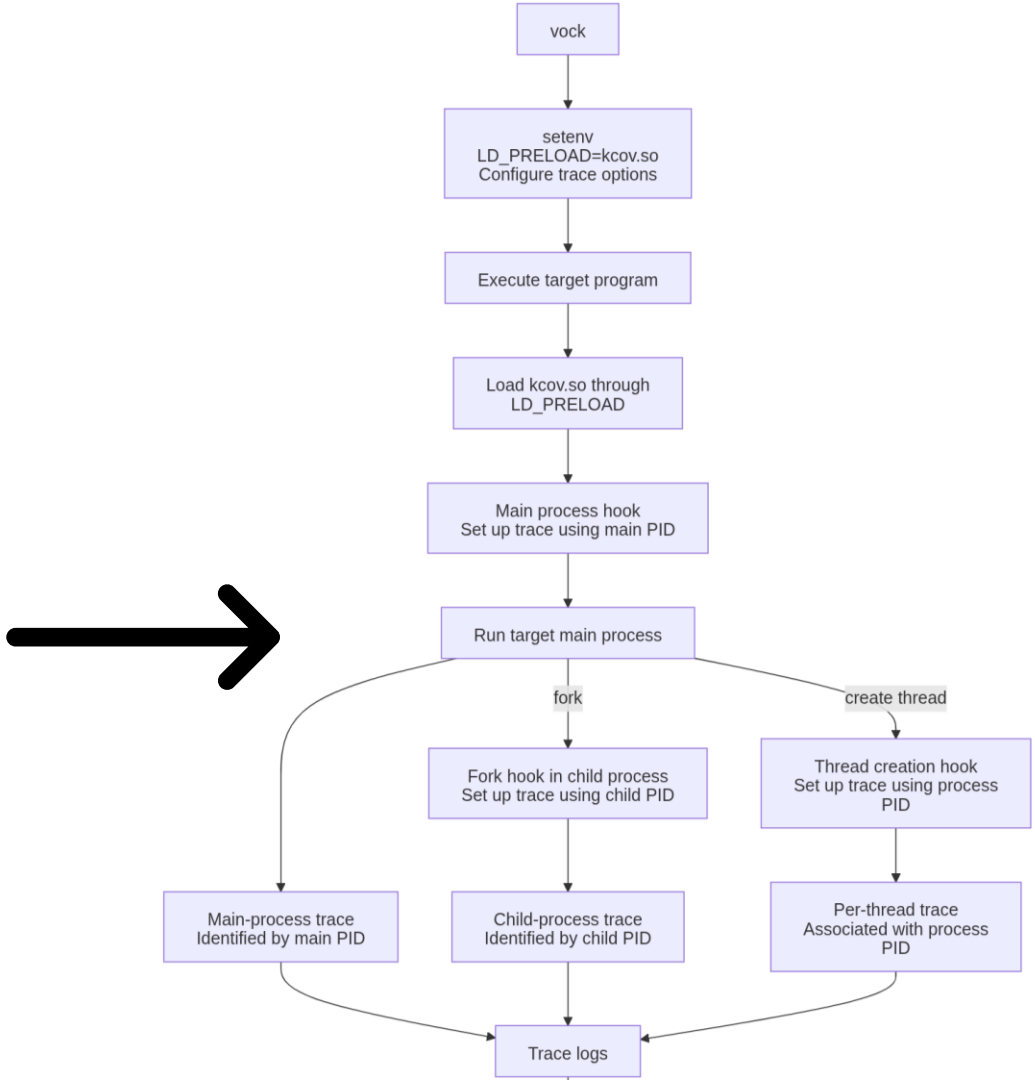
```
... @@ ksmbd_conn_get
218 |  * if @conn is NULL.
219 |  */
220 | struct ksmbd_conn *ksmbd_conn_get(struct ksmbd_conn *conn)
221 > {
222 |         if (!conn)
223 |                 return NULL;
224 |
225 >         atomic_inc(&conn->refcnt);
226 |         return conn;
227 | }
228 |
... @@ ksmbd_conn_put
239 |  * non-NULL while a conn reference is held.
240 |  */
241 | void ksmbd_conn_put(struct ksmbd_conn *conn)
242 > {
243 |         if (!conn)
244 |                 return;
245 |
246 >         if (atomic_dec_and_test(&conn->refcnt))
247 |                 queue_work(ksmbd_conn_wq, &conn->release_work);
248 | }
249 |
... @@ ksmbd_conn_lookup_dialect
343 | }
344 |
345 | bool ksmbd_conn_lookup_dialect(struct ksmbd_conn *c)
346 > {
347 |         struct ksmbd_conn *t;
348 |         int bkt;
349 |         bool ret = false;
350 |
351 |         down_read(&conn_list_lock);
352 >         hash_for_each(conn_list, bkt, t, hlist) {
353 >                 if (memcmp(t->ClientGUID, c->ClientGUID, SMB2_CLIENT_GUID_SIZE))
354 |                         continue;
355 |
356 |                 ret = true;
... @@ ksmbd_conn_enqueue_request
361 | }
362 |
363 | void ksmbd_conn_enqueue_request(struct ksmbd_work *work)
364 > {
365 |         struct ksmbd_conn *conn = work->conn;
366 |         struct list_head *requests_queue = NULL;
367 |
```

fs/smb/server/vfs.c (37 lines)

```
... @@ ksmbd_vfs_path_lookup
 61 |         int err;
 62 |         struct dentry *d;
 63 |
 64 >         if (pathname[0] == '\0') {
 65 |                 pathname = share_conf->path;
 66 |                 root_share_path = NULL;
 67 |         } else {
... @@ ksmbd_vfs_path_lookup
 96 |
 97 |         d = lookup_noperm_unlocked(&last, path->dentry);
 98 |         if (!IS_ERR(d) && d_is_negative(d)) {
 99 >                 dput(d);
100 |                 d = ERR_PTR(-ENOENT);
101 |         }
102 |         if (IS_ERR(d)) {
... @@ ksmbd_vfs_path_lookup
108 |
109 |         if (test_share_config_flag(share_conf, KSMBD_SHARE_FLAG_CROSSMNT)) {
110 |                 err = follow_down(path, 0);
111 >                 if (err < 0) {
112 |                         path_put(path);
113 |                         return -ENOENT;
114 |                 }
... @@ ksmbd_vfs_create
146 |  * Return:      0 on success, otherwise error
147 |  */
148 | int ksmbd_vfs_create(struct ksmbd_work *work, const char *name, umode_t mode)
149 > {
150 |         struct path path;
151 |         struct dentry *dentry;
152 |         int err;
... @@ ksmbd_vfs_write
467 | int ksmbd_vfs_write(struct ksmbd_work *work, struct ksmbd_file *fp,
468 |                     char *buf, size_t count, loff_t *pos, bool sync,
469 |                     ssize_t *written)
470 > {
471 |         struct file *filp;
472 |         loff_t  offset = *pos;
473 |         int err = 0;
474 |
475 |         if (work->conn->connection_type) {
476 |                 if (!(fp->daccess & (FILE_WRITE_DATA_LE | FILE_APPEND_DATA_LE)) ||
477 >                     (S_ISDIR(file_inode(fp->filp)->i_mode) &&
478 |                      !ksmbd_stream_fd(fp))) {
479 |                         pr_err("no right to write(%pD)\n", fp->filp);
480 |                         err = -EACCES;
... @@ ksmbd_vfs_write
491 |                 goto out;
492 |         }
493 |
494 >         if (!work->tcon->posix_extensions) {
495 |                 err = check_lock_range(filp, *pos, *pos + count - 1, WRITE);
496 |                 if (err) {
497 |                         pr_err("unable to write due to lock\n");
```

vock-smbclient-tcp-worker.log

```
# vock --mode kcov --kernel-src . --vmlinux vmlinux \
  smbclient "//$TCP_HOST/$SHARE_NAME" -N -c "put $file $base"
```


vock
setenv
LD_PRELOAD=kcov.so
Configure trace options
Execute target program
Load kcov.so through
LD_PRELOAD
Main process hook
Set up trace using main PID
Run target main process
fork
create thread
Fork hook in child process
Set up trace using child PID
Thread creation hook
Set up trace using process
PID
Main-process trace
Identified by main PID
Child-process trace
Identified by child PID
Per-thread trace
Associated with process
PID
Trace logs

```
kcov[219]: coverage enabled
kcov[222]: coverage enabled
kcov[223]: coverage enabled
kcov[222]: 165610 PCs → local-222.log
kcov[223]: 164347 PCs → local-223.log
kcov[219]: 332580 PCs → local-219.log
[vock] coverage-219.html: rendering the first 20000 of 332580 trace entries (VOCK_ORDERED_HTML_ROWS=0 for all)
[vock] local-219.log → coverage-219.html
[vock] coverage-222.html: rendering the first 20000 of 165610 trace entries (VOCK_ORDERED_HTML_ROWS=0 for all)
[vock] local-222.log → coverage-222.html
[vock] coverage-223.html: rendering the first 20000 of 164347 trace entries (VOCK_ORDERED_HTML_ROWS=0 for all)
[vock] local-223.log → coverage-223.html
```



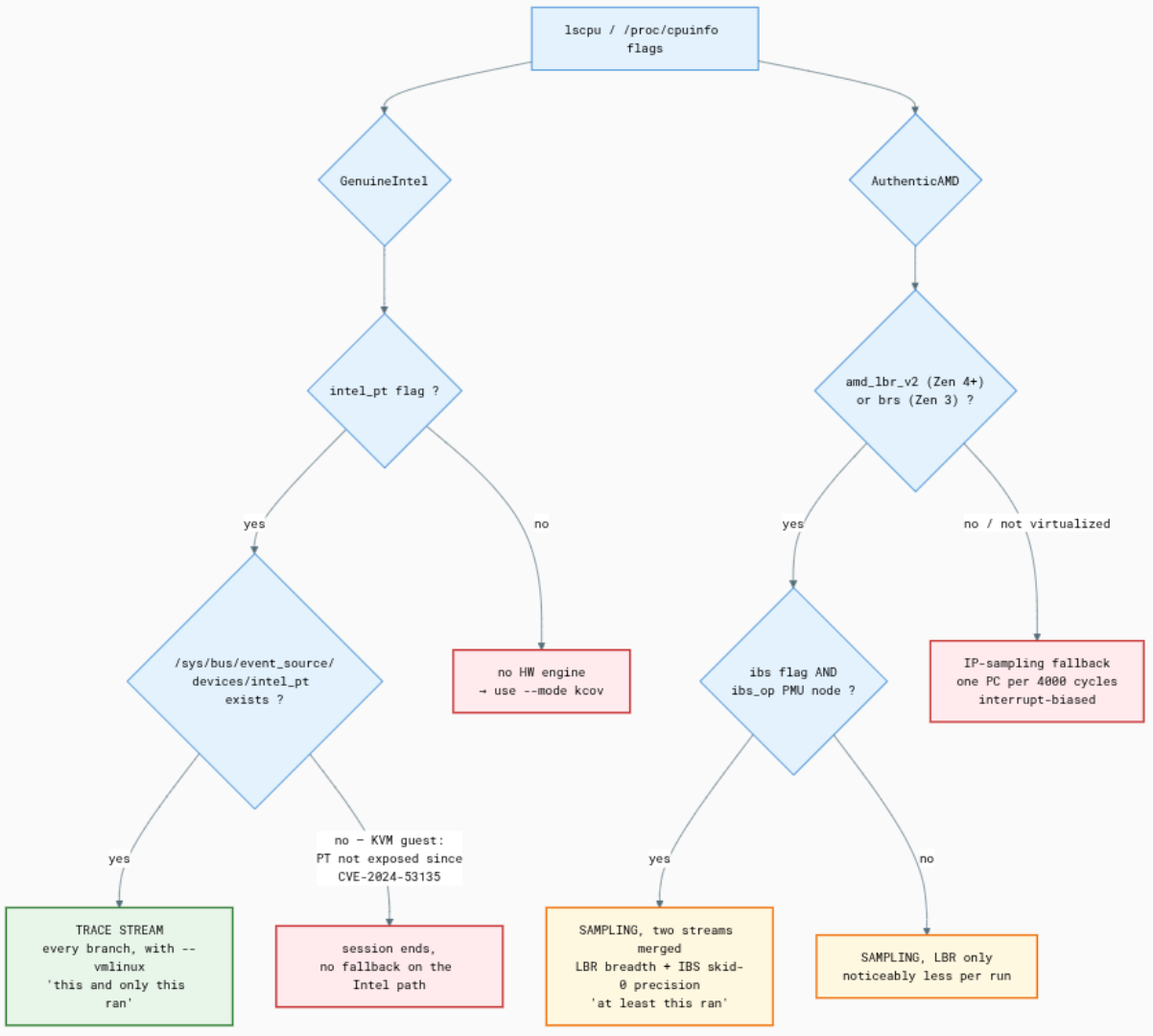
Motivation: How Can We Collect Kernel Coverage Without Enabling KCOV?

Motivation: How Can We Collect Kernel Coverage Without Enabling KCOV?

- Use **--mode hw** when you want branch-exact completeness on one task, proof that code did not run, or coverage on a kernel built without CONFIG_KCOV.
  - For normal user needs kernel.perf_event_paranoid needs ≤ -1
- Use --mode kcov
  - Coverage across threads, forks from user process
  - Remote kernel context coverage collecting
  - KVM guest

Supporting to Hardware mode

- Intel Processor Trace Stream
  - /sys/bus/event_source/devices/intel_pt/type
- AMD Instruction-Based Sampling
  - /sys/bus/event_source/devices/ibs_op/type
- Arm CoreSight ETM Trace Stream
  - Currently, not supported in vock
  - /sys/bus/event_source/devices/cs_etm/type


lscpu / /proc/cpuinfo
flags
GenuineIntel
AuthenticAMD
intel_pt flag ?
amd_lbr_v2 (Zen 4+)
or brs (Zen 3) ?
yes
no
yes
no / not virtualized
/sys/bus/event_source/
devices/intel_pt
exists ?
no HW engine
→ use --mode kcov
ibs flag AND
ibs_op PMU node ?
IP-sampling fallback
one PC per 4000 cycles
interrupt-biased
yes
no – KVM guest:
PT not exposed since
CVE-2024-53135
yes
no
TRACE STREAM
every branch, with --
vmlinux
'this and only this
ran'
session ends,
no fallback on the
Intel path
SAMPLING, two streams
merged
LBR breadth + IBS skid-
0 precision
'at least this ran'
SAMPLING, LBR only
noticeably less per run

Supporting to Hardware mode

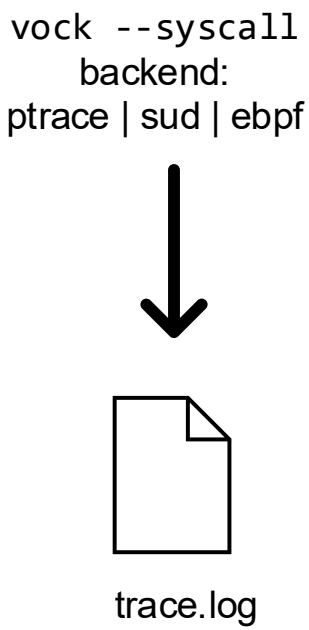
- Here is the output from the vock with `cat /proc/zoneinfo`
  - KCOV: https://gist.github.com/yskzalloc/c1197acba2a5423bc2066798b3cddbb0#file-kcov-log
  - Intel Core ULTRA 7: https://gist.github.com/yskzalloc/c1197acba2a5423bc2066798b3cddbb0#file-intel-pt-log
  - AMD EPYC 9654: https://gist.github.com/yskzalloc/c1197acba2a5423bc2066798b3cddbb0#file-amd-lbr-log

	Intel PT	AMD LBR + IBS (merged)
Class	trace stream	sampling
Completeness	every branch	~10% of touched PCs per run, hot-path biased
Can prove code did **not** run	yes (within the window)	**never** — absence means nothing
Execution ordering	full ordered path	16-branch islands + isolated points
Per-record precision	branch-exact	IBS: skid-0 retired ops; LBR: exact branch pairs
Overhead / data	~2–15%, 100s of MB	near zero, KBs
Decode cost	heavy (needs exact code bytes)	trivial
Works in KVM guests	no (disabled after CVE-2024-53135)	no (falls back to IP sampling)


Syscall Tracing

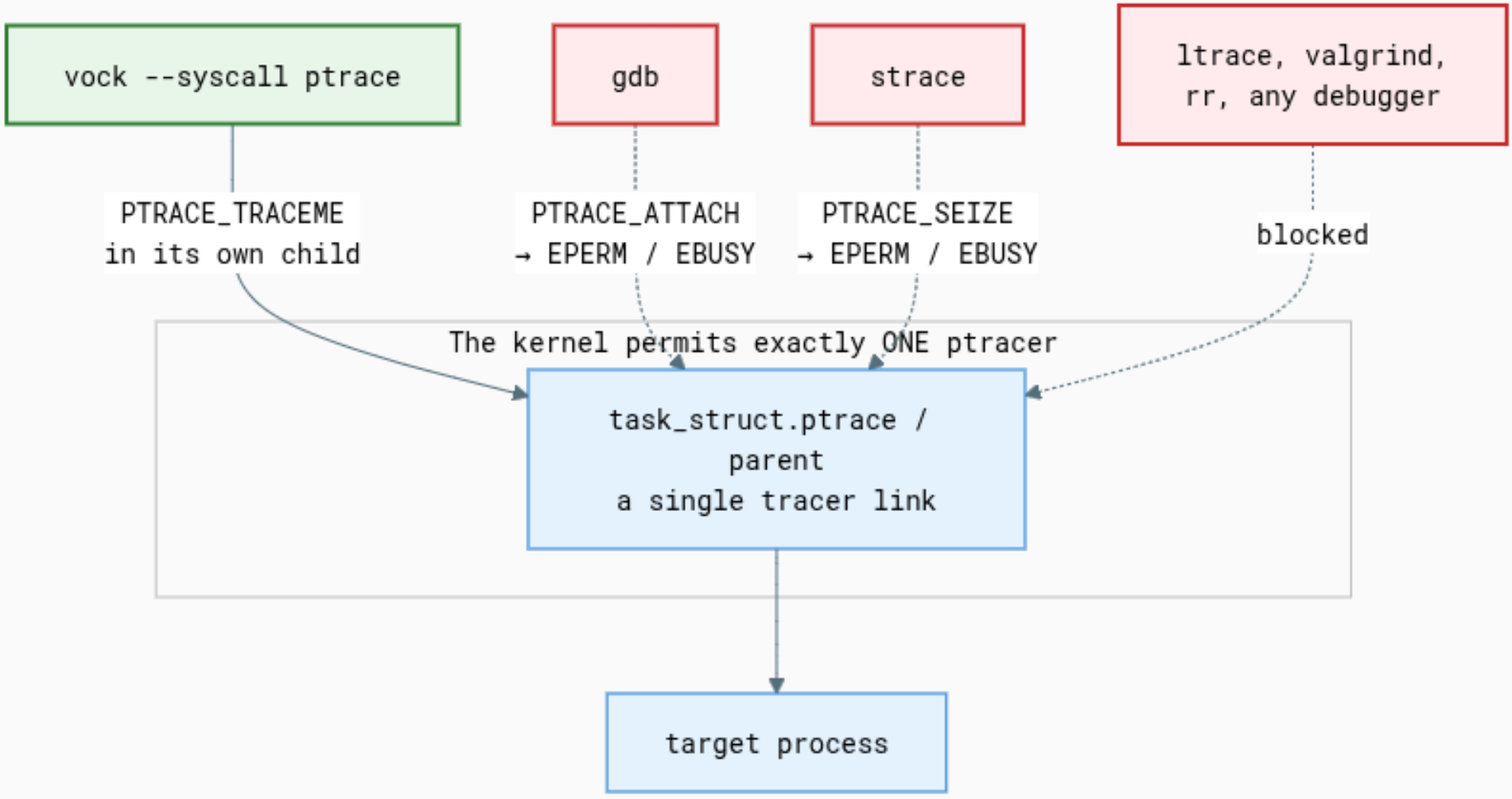
Supporting 3 Syscall tracing Modes

- ptrace
- Syscall User Dispatch
- eBPF raw_syscalls tracepoints



ptrace

- Uses classic PTRACE_TRACEME on child. Works unprivileged on hardened systems and survives execve.
- Only one ptracer allowed per process.
- Conflicts directly with gdb, strace, or rr debugging sessions.


vock --syscall ptrace
gdb
strace
ltrace, valgrind,
rr, any debugger
PTRACE_TRACEME
in its own child
PTRACE_ATTACH
→ EPERM / EBUSY
PTRACE_SEIZE
→ EPERM / EBUSY
blocked
The kernel permits exactly ONE ptracer
task_struct.ptrace /
parent
a single tracer link
target process

Syscall User Dispatch: In-Process Trapping

- Kernel ≥ 5.11 with CONFIG_SYSCALL_USER_DISPATCH=y
- vock only support it on x86_64
- Traps syscalls to SIGSYS handler when selector byte equals BLOCK.
- Custom sa_restorer trampoline in allow-window prevents infinite recursion during sigreturn.
- Need `echo 0 | sudo tee /proc/sys/vm/mmap_min_addr`

eBPF: Hand-Assembled Bytecode

- CONFIG_BPF_SYSCALL=y
- For normal user CAP_BPF+CAP_PERFMON unprivileged_bpf_disabled=0 or root
- Encodes raw struct bpf_insn 64-bit words directly without libbpf or clang dependencies.
- Attaches to `raw_syscalls:sys_enter` and `sys_exit tracepoints` with thread-group ID filtering.
- Emits 80-byte records to 256 KB ring buffer without interrupting target thread execution.
- Completely compatible alongside active gdb or rr debugging sessions.

Metric / Feature	**ptrace (Default)**	Syscall User Dispatch	eBPF
Privilege Needed	None (own child)	None	CAP_BPF + CAP_PERFMON
Debugger Coexistence	No (Exclusive)	Yes	Yes (GDB Compatible)
Survives execve	Yes	No (Stops at exec)	Yes



Additional Features for Security Research: Built-in DWARF reader, syzlang Interpretation and more

Why addr2line is slow

1. It indexes everything. Builds tables for all compilation units, whether or not your addresses touch them
2. It caches everything. GNU addr2line peaks near 1 GiB RSS and gets OOM-killed in a small VM guest
3. Cold pages in a VM. Inside a guest, the first touch of every vmlinux page is a round trip to the host, and it is one trip at a time

vock's answer: read it ourselves, and read less

- Do mmap the vmlinux read-only.
  - every section is just a slice of that mapping
- Build the address index from .debug_aranges alone
  - one small sequential read
- Parse a unit's line table and function table only when an address lands in it
- Automatic De-KASLR

vock's answer: stream pipeline

- Pre-warm: start before the program finishes
  - The DWARF load starts on a background thread while the traced program is still running
- The stream pipeline: three threads, nothing batched

Reader thread → **Main thread** → **Writer thread**

4 MiB blocks | dedup + symbolize | srccov.log

(per-TID logs) | (only fresh PCs) | (streams as blocks complete)

KCOV addr DWARF Reader	Symbolization Time (8.4K Unique PCs)
🥇 vock (in-process)	~0.3 s (index ready at 80 ms)
🥈 llvm-addr2line	0.6 s
🥉 GNU addr2line	9 s

[Ongoing Work] vock execprog: A syzlang Interpreter

- Goal
  - Drop-in compatible with syzkaller tooling
  - Capture with `vock --syzlang`, replay with `vock execprog -cover`, and you get per-call kernel coverage for a reproducer you did not have to modify.
- syzlang interpretation works success on selftest one case
  - It reproduces a real KASAN use-after-free

[Ongoing Work] Integrating vock with KCOV-dataflow

- Capture all function arguments, return values, and kernel coverage for the target user process.
- RFCs for Both LLVM and the Linux Kernel
  - LLVM RFC for `-fsanitize-coverage=trace-args/ret`: https://discourse.llvm.org/t/rfc-sanitizercoverage-add-fsanitize-coverage-trace-args-trace-ret/91026
  - LPC 26 for KCOV dataflow: https://lpc.events/event/20/contributions/2402/

```
do_el0_svc({0x4, 0x4, 0xffffffff, 0x0, 0x0, 0x0})
0x0 = B...
sb_start_write()
file_start_write()
0x0 = _RNvCsdfZGIOKgjaD_22eight_struct_args_rust13write_handler()
  0x11 = rsf_1(0x11) [eight_struct_args_rust]
  0x33 = rsf_2(0x11, {0x11, 0x22}) [eight_struct_args_rust]
```

Q & A



vock: Mapping Kernel Attack Surface with Zero-Configuration Coverage


LINUX
SECURITY
SUMMIT
EUROPE

References

- vock: https://github.com/yskzalloc/vock
- uftrace: https://github.com/namhyung/uftrace
- strace: https://github.com/strace/strace
- lazypolin: https://github.com/lazypoline/lazypoline.git
- ebpftracer: https://github.com/KnightChaser/ebpftracer
- libbpf: https://github.com/libbpf/libbpf
- gimli: https://github.com/gimli-rs/gimli