author	Eduard Zingerman <eddyz87@gmail.com>	2024-12-09 20:10:55 -0800
committer	Alexei Starovoitov <ast@kernel.org>	2024-12-10 10:24:57 -0800
commit	51081a3f25c742da5a659d7fc6fd77ebfdd555be (patch)	
tree	c2e6ddf855e44e677db87c7dc23d2263a029b935	
parent	b238e187b4a2d3b54d80aec05a9cab6466b79dde (diff)	
download	linux-51081a3f25c742da5a659d7fc6fd77ebfdd555be.tar.gz	

bpf: track changes_pkt_data property for global functions

```
When processing calls to certain helpers, verifier invalidates all
packet pointers in a current state. For example, consider the
following program:

    __attribute__((__noinline__))
    long skb_pull_data(struct __sk_buff *sk, __u32 len)
    {
        return bpf_skb_pull_data(sk, len);
    }

    SEC("tc")
    int test_invalidate_checks(struct __sk_buff *sk)
    {
        int *p = (void *)(long)sk->data;
        if ((void *)(p + 1) > (void *)(long)sk->data_end) return TCX_DROP;
        skb_pull_data(sk, 0);
        *p = 42;
        return TCX_PASS;
    }

After a call to bpf_skb_pull_data() the pointer 'p' can't be used
safely. See function filter.c:bpf_helper_changes_pkt_data() for a list
of such helpers.

At the moment verifier invalidates packet pointers when processing
helper function calls, and does not traverse global sub-programs when
processing calls to global sub-programs. This means that calls to
helpers done from global sub-programs do not invalidate pointers in
the caller state. E.g. the program above is unsafe, but is not
rejected by verifier.

This commit fixes the omission by computing field
bpf_subprog_info->changes_pkt_data for each sub-program before main
verification pass.
changes_pkt_data should be set if:
- subprogram calls helper for which bpf_helper_changes_pkt_data
  returns true;
- subprogram calls a global function,
  for which bpf_subprog_info->changes_pkt_data should be set.

The verifier.c:check_cfg() pass is modified to compute this
information. The commit relies on depth first instruction traversal
done by check_cfg() and absence of recursive function calls:
- check_cfg() would eventually visit every call to subprogram S in a
  state when S is fully explored;
```

```
- when S is fully explored:
  - every direct helper call within S is explored
    (and thus changes_pkt_data is set if needed);
  - every call to subprogram S1 called by S was visited with S1 fully
    explored (and thus S inherits changes_pkt_data from S1).

The downside of such approach is that dead code elimination is not
taken into account: if a helper call inside global function is dead
because of current configuration, verifier would conservatively assume
that the call occurs for the purpose of the changes_pkt_data
computation.

Reported-by: Nick Zavaritsky <mejedi@gmail.com>
Closes: https://lore.kernel.org/bpf/0498CA22-5779-4767-9C0C-A9515CEA711F@gmail.com/
Signed-off-by: Eduard Zingerman <eddyz87@gmail.com>
Link: https://lore.kernel.org/r/20241210041100.1898468-4-eddyz87@gmail.com
Signed-off-by: Alexei Starovoitov <ast@kernel.org>
```

Diffstat

-rw-r--r--	include/linux/bpf_verifier.h	1
-rw-r--r--	kernel/bpf/verifier.c	32

2 files changed, 32 insertions, 1 deletions

```
diff --git a/include/linux/bpf_verifier.h b/include/linux/bpf_verifier.h
index f4290c179bee0a..48b7b2eeb7e21f 100644
--- a/include/linux/bpf_verifier.h
+++ b/include/linux/bpf_verifier.h
@@ -659,6 +659,7 @@ struct bpf_subprog_info {
        bool args_cached: 1;
        /* true if bpf_fastcall stack region is used by functions that can't be inlined */
        bool keep_fastcall_stack: 1;
+       bool changes_pkt_data: 1;

        enum priv_stack_mode priv_stack_mode;
        u8 arg_cnt;

diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index ad3f6d28e8e4ff..6a29b68cebd6f3 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -10042,6 +10042,8 @@ static int check_func_call(struct bpf_verifier_env *env, struct bpf_insn *insn,

                verbose(env, "Func#%d ('%s') is global and assumed valid.\n",
                        subprog, sub_name);
+               if (env->subprog_info[subprog].changes_pkt_data)
+                       clear_all_pkt_pointers(env);
                /* mark global subprog for verifying after main prog */
                subprog_aux(env, subprog)->called = true;
                clear_caller_saved_regs(env, caller->regs);
@@ -16246,6 +16248,29 @@ enforce_retval:
        return 0;
 }

+static void mark_subprog_changes_pkt_data(struct bpf_verifier_env *env, int off)
+{
+       struct bpf_subprog_info *subprog;
+
+       subprog = find_containing_subprog(env, off);
+       subprog->changes_pkt_data = true;
+}
+
+/* 't' is an index of a call-site.
+ * 'w' is a callee entry point.
+ * Eventually this function would be called when env->cfg.insn_state[w] == EXPLORED.
+ * Rely on DFS traversal order and absence of recursive calls to guarantee that
```

```
+ * callee's change_pkt_data marks would be correct at that moment.
+ */
+static void merge_callee_effects(struct bpf_verifier_env *env, int t, int w)
+{
+       struct bpf_subprog_info *caller, *callee;
+
+       caller = find_containing_subprog(env, t);
+       callee = find_containing_subprog(env, w);
+       caller->changes_pkt_data |= callee->changes_pkt_data;
+}
+
 /* non-recursive DFS pseudo code
  * 1  procedure DFS-iterative(G,v):
  * 2      label v as discovered
@@ -16379,6 +16404,7 @@ static int visit_func_call_insn(int t, struct bpf_insn *insns,
                                bool visit_callee)
 {
        int ret, insn_sz;
+       int w;

        insn_sz = bpf_is_ldimm64(&insns[t]) ? 2 : 1;
        ret = push_insn(t, t + insn_sz, FALLTHROUGH, env);
@@ -16390,8 +16416,10 @@ static int visit_func_call_insn(int t, struct bpf_insn *insns,
        mark_jmp_point(env, t + insn_sz);

        if (visit_callee) {
+               w = t + insns[t].imm + 1;
                mark_prune_point(env, t);
-               ret = push_insn(t, t + insns[t].imm + 1, BRANCH, env);
+               merge_callee_effects(env, t, w);
+               ret = push_insn(t, w, BRANCH, env);
        }
        return ret;
 }
@@ -16708,6 +16736,8 @@ static int visit_insn(int t, struct bpf_verifier_env *env)
                        mark_prune_point(env, t);
                        mark_jmp_point(env, t);
                }
+               if (bpf_helper_call(insn) && bpf_helper_changes_pkt_data(insn->imm))
+                       mark_subprog_changes_pkt_data(env, t);
                if (insn->src_reg == BPF_PSEUDO_KFUNC_CALL) {
                        struct bpf_kfunc_call_arg_meta meta;
```