Linux kernel: CVE-2022-1015,CVE-2022-1016 in nf_tables cause privilege escalation, information leak

---

From: David Bouman <davidbouman35 () gmail com>
Date: Mon, 28 Mar 2022 20:28:21 +0200

Hello list,

I'm reporting two linux kernel vulnerabilities in the nf_tables component of the netfilter subsystem that I found.

CVE-2022-1015 pertains to an out of bounds access in nf_tables expression evaluation due to validation of user register indices. It leads to local privilege escalation, for example by overwriting a stack return address OOB with a crafted nft_expr_payload.

CVE-2022-1015 is exploitable starting from commit 345023b0db3 ("netfilter: nftables: add nft_parse_register_store() and use it"), v5.12 and has been fixed in commit 6e1acfa387b9 ("netfilter: nf_tables: validate registers coming from userspace.").

The bug has been present since commit 49499c3e6e18 ("netfilter: nf_tables: switch registers to 32 bit addressing"), but to my knowledge has not been exploitable until v5.12.

CVE-2022-1016 pertains to uninitialized stack data in the nft_do_chain routine. CVE-2022-1016 is exploitable starting from commit 96518518cc41 (original merge of nf_tables), v3.13-rc1, and has been fixed in commit 4c905f6740a3 ("netfilter: nf_tables: initialize registers in nft_do_chain()").

I will be releasing a detailed blog post and exploit code for both vulnerabilities in a few days.

Root cause CVE-2022-1016: (it is the shortest, so I will begin with it)

The nft_do_chain routine in net/netfilter/nf_tables_core.c does not initialize the register data that nf_tables expressions can read from- and write to. These expressions inherently exhibit side effects that can be used to determine the register data, which can contain kernel image pointers, module pointers, and allocation pointers depending on the code path taken to end up at nft_do_chain.

```
unsigned int
nft_do_chain(struct nft_pktinfo *pkt, void *priv)
```

```
{
        const struct nft_chain *chain = priv, *basechain = chain;
        const struct net *net = nft_net(pkt);
        struct nft_rule *const *rules;
        const struct nft_rule *rule;
        const struct nft_expr *expr, *last;
        struct nft_regs regs; // <-------- VULNERABLE! NOT INITIALIZED.
        unsigned int stackptr = 0;
        struct nft_jumpstack jumpstack[NFT_JUMP_STACK_SIZE];
        bool genbit = READ_ONCE(net->nft.gencursor);
        struct nft_traceinfo info;

        info.trace = false;
        if (static_branch_unlikely(&nft_trace_enabled))
                nft_trace_init(&info, pkt, &regs.verdict, basechain);
do_chain:
        if (genbit)
                rules = rcu_dereference(chain->rules_gen_1);
        else
                rules = rcu_dereference(chain->rules_gen_0);

next_rule:
        rule = *rules;
        regs.verdict.code = NFT_CONTINUE;
        for (; *rules ; rules++) {
                rule = *rules;
                nft_rule_for_each_expr(expr, last, rule) {
                        if (expr->ops == &nft_cmp_fast_ops)
                                nft_cmp_fast_eval(expr, &regs);
                        else if (expr->ops == &nft_bitwise_fast_ops)
                                nft_bitwise_fast_eval(expr, &regs);
                        else if (expr->ops != &nft_payload_fast_ops ||
                                 !nft_payload_fast_eval(expr, &regs, pkt))
                                expr_call_ops_eval(expr, &regs, pkt);
                                ...
```

Root cause CVE-2022-1015:

(below is pasted from my original security () kernel org report)

Hello, I'm mailing to report a vulnerability I found in nf_tables component of the netfilter subsystem. The vulnerability gives an attacker a powerful primitive that can be used to both read from and write to relative stack data. This can lead to arbitrary code execution by an attacker.

In order for an unprivileged attacker to exploit this issue, unprivileged user- and network namespaces access is required (CLONE_NEWUSER | CLONE_NEWNET). The bug relies on a compiler optimization that introduces behavior that the maintainer did not account for, and most likely only occurs on kernels with `CONFIG_CC_OPTIMIZE_FOR_PERFORMANCE=y`. I successfully exploited the bug on x86_64 kernel version 5.16-rc3, but I believe this vulnerability exists across different kernel versions and architectures (more on this later).

Without further ado:

The bug resides in `linux/net/netfilter/nf_tables_api.c`, in the `nft_validate_register_store` and `nft_validate_register_load` routines. These routines are used to check if nft expression parameters supplied by the user are sound and won't cause OOB stack accesses when evaluating the expression.

From my 5.16-rc3 kernel source (d58071a8a76d779eedab38033ae4c821c30295a5: Linux 5.16-rc3):

nft_validate_register_store:

```
static int nft_validate_register_store(const struct nft_ctx *ctx,
      enum nft_registers reg,
      const struct nft_data *data,
      enum nft_data_types type,
      unsigned int len)
{
int err;

switch (reg) {
        ...
default:
if (reg < NFT_REG_1 * NFT_REG_SIZE / NFT_REG32_SIZE)
return -EINVAL;
if (len == 0)
return -EINVAL;
if (reg * NFT_REG32_SIZE + len >
   sizeof_field(struct nft_regs, data))
return -ERANGE;

if (data != NULL && type != NFT_DATA_VALUE)
return -EINVAL;
return 0;
}
}
```

nft_validate_register_load:

```
static int nft_validate_register_load(enum nft_registers reg, unsigned int len)
{
if (reg < NFT_REG_1 * NFT_REG_SIZE / NFT_REG32_SIZE)
return -EINVAL;
if (len == 0)
return -EINVAL;
if (reg * NFT_REG32_SIZE + len > sizeof_field(struct nft_regs, data))
return -ERANGE;

return 0;
}
```

The problem lies in the fact that `enum nft_registers reg` is not guaranteed only be a single byte. As per the C89 specification, 3.1.3.3 Enumeration constants: `An identifier declared as an enumeration constant has type int.`.

Effectively this implies that the compiler is free to emit code that operates on `reg` as if it were a 32-bit value. If this is the case (and it is on the kernel I tested), a user can forge an expression register value that will overflow upon multiplication with `NFT_REG32_SIZE` (4) and upon addition with `len`, will be a value smaller than `sizeof_field(struct nft_regs, data)` (0x50). Once this check passes, the least significant byte of `reg` can still contain a value that will index outside of the bounds of the `struct nft_regs regs` that it will later be used with.

Take for example a `reg` value of `0xfffffff8` and a `len` value of `0x40`. The expression `reg * 4 + len` will then result in `0xffffffe0 + 0x40 = 0x20`, which is lower than `0x50`. This makes that a value of `0xf8` is recognized as a valid index, and is subsequently assigned to a register value in the expression info structs.

Here is a snippet of the x86_64 assembly code that these functions might generate:

```
Disassembly of section .text:

0000000000002ed0 <nft_validate_register_store>:
2ed0: e8 00 00 00 00 callq 2ed5 <nft_validate_register_store+0x5>
    2ed5: 55                   push   %rbp
    2ed6: 48 89 e5             mov    %rsp,%rbp
    2ed9: 41 54                 push   %r12
    2edb: 85 f6                 test   %esi,%esi
2edd: 75 2b jne 2f0a <nft_validate_register_store+0x3a>
    2edf: 81 f9 00 ff ff ff     cmp    $0xffffff00,%ecx
2ee5: 75 49 jne 2f30 <nft_validate_register_store+0x60>
    2ee7: 45 31 e4             xor    %r12d,%r12d
    2eea: 48 85 d2             test   %rdx,%rdx
2eed: 74 3a je 2f29 <nft_validate_register_store+0x59>
    2eef: 8b 02                 mov    (%rdx),%eax
    2ef1: 83 c0 04             add    $0x4,%eax
    2ef4: 83 f8 01             cmp    $0x1,%eax
2ef7: 77 30 ja 2f29 <nft_validate_register_store+0x59>
    2ef9: 48 8b 72 08           mov    0x8(%rdx),%rsi
    2efd: e8 7e da ff ff       callq  980 <nf_tables_check_loops>
    2f02: 85 c0                 test   %eax,%eax
    2f04: 44 0f 4e e0           cmovle %eax,%r12d
2f08: eb 1f jmp 2f29 <nft_validate_register_store+0x59>
    2f0a: 83 fe 03             cmp    $0x3,%esi
2f0d: 76 21 jbe 2f30 <nft_validate_register_store+0x60>
    2f0f: 45 85 c0             test   %r8d,%r8d
2f12: 74 1c je 2f30 <nft_validate_register_store+0x60>
    2f14: 41 8d 04 b0           lea    (%r8,%rsi,4),%eax
    2f18: 83 f8 50             cmp    $0x50,%eax
2f1b: 77 1b ja 2f38 <nft_validate_register_store+0x68>
    2f1d: 48 85 d2             test   %rdx,%rdx
2f20: 74 04 je 2f26 <nft_validate_register_store+0x56>
    2f22: 85 c9                 test   %ecx,%ecx
2f24: 75 0a jne 2f30 <nft_validate_register_store+0x60>
    2f26: 45 31 e4             xor    %r12d,%r12d
    2f29: 44 89 e0             mov    %r12d,%eax
    2f2c: 41 5c                 pop    %r12
    2f2e: 5d                   pop    %rbp
    2f2f: c3                   retq
    2f30: 41 bc ea ff ff ff     mov    $0xffffffea,%r12d
2f36: eb f1 jmp 2f29 <nft_validate_register_store+0x59>
    2f38: 41 bc de ff ff ff     mov    $0xffffffde,%r12d
2f3e: eb e9 jmp 2f29 <nft_validate_register_store+0x59>
```

the `lea` instruction at `2f14` will multiply `%rsi` (reg) by 4 and add `%r8` len to it.

I created a working local privilege escalation exploit by using such an out of bounds index to copy stack data to the actual register area (declared in nf_tables_core.c:nft_do_chain). Then, I wrote a a few nft rules that drop or accept packets depending on whether the targeted byte is greater than the constant comparand in the rule or not. This way I could create a binary search procedure that could determine the value of the leaked byte by registering whether the packet was dropped or not. This results in a kernel address leak.

```
Finally, I used a nft payload expression to write my arbitrary data supplied in a packet to
the stack in order to overwrite a return address and execute a ROP chain.

An alternative exploitation strategy would be to overwrite to verdict register (including
its chain pointer) to arbitrary values, as you can now get an register index of 0 in the
same manner.

------------------------------

David Bouman
```

Current thread:

Linux kernel: CVE-2022-1015,CVE-2022-1016 in nf_tables cause privilege escalation, information leak ***David Bouman (Mar 28)***