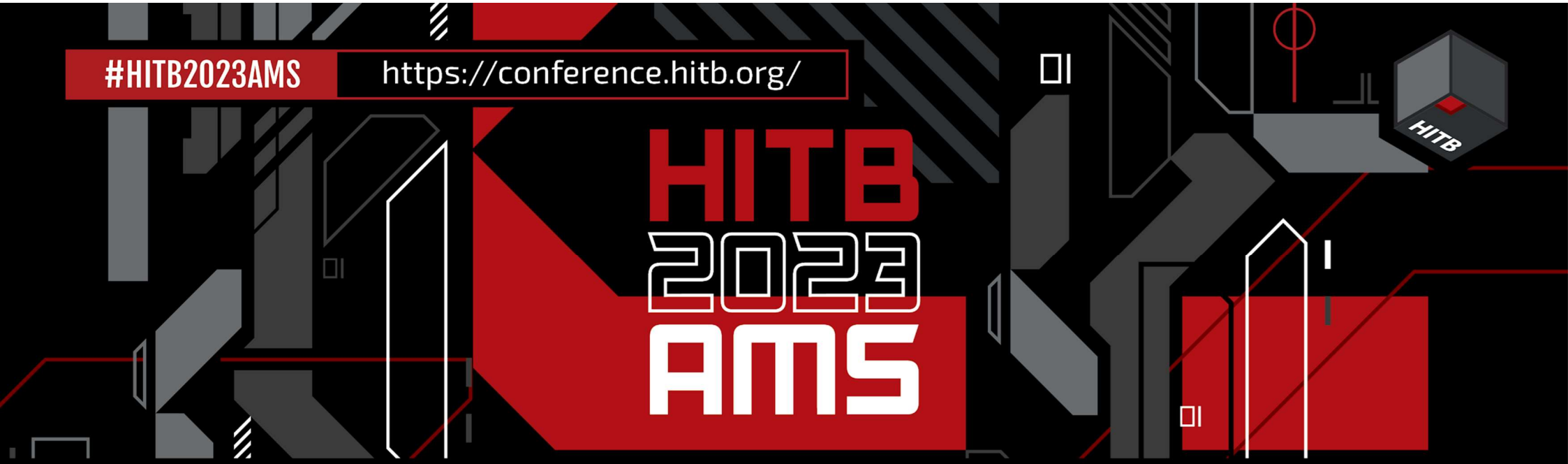


The Return of Stack Overflows in the Linux Kernel

Davide Ornaghi | Offensive Security Specialist | Betrusted

Davide Ornaghi – @TurtleARM97

- Offensive Security Specialist at Betrusted
- Instructor for malware analysis and penetration testing seminars
- MSc in IT Security, CEH Master, OSCP, OSWP
- 0-day enjoyer

Betrusted

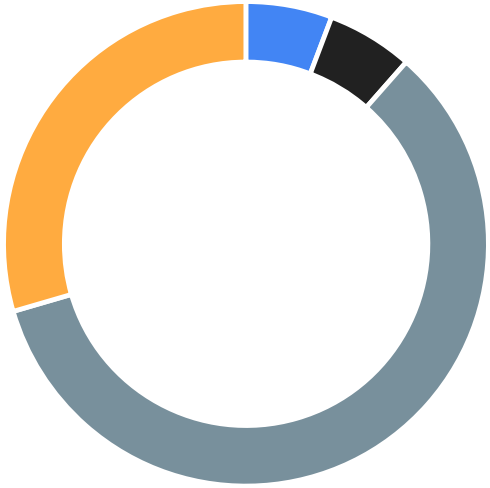
Agenda

- Trends in Linux 0-days
  - Vulnerability classes and components
- Analysis of recent kernel vulns
  - Main security mitigations
- The anatomy of a successful Stack Overflow
  - Bypassing security controls
  - Leaving the interrupt context
- Mitigating the risk inside our kernels

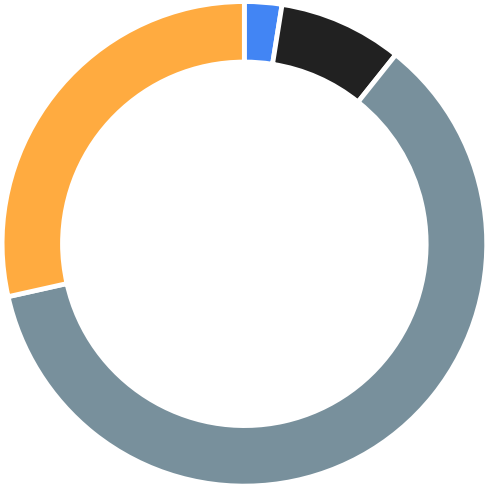
Trends in Linux 0-days

Vulnerability distribution by class

Linux 3.x and 4.x



Linux 5.x and 6.x

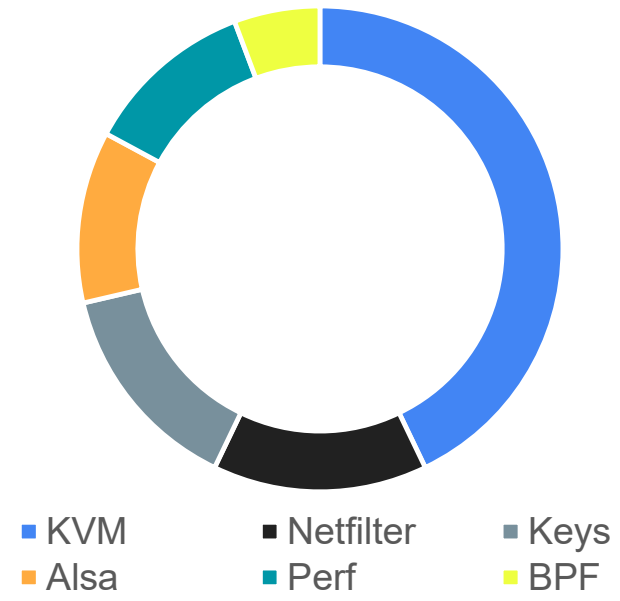


Vulnerability distribution by class – source?

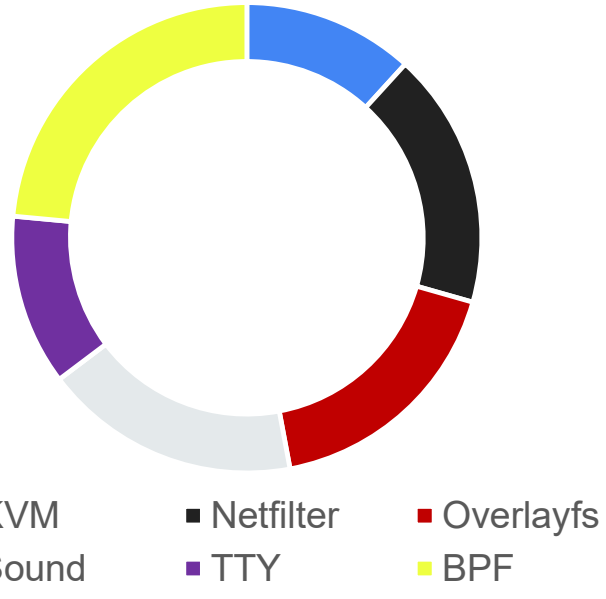
```
┌──(kali㉿kali)-[~/hitb]
└─$ git clone https://github.com/nluedtke/linux_kernel_cves.git
Cloning into 'linux_kernel_cves'...
remote: Enumerating objects: 59915, done.
remote: Counting objects: 100% (18311/18311), done.
remote: Compressing objects: 100% (6105/6105), done.
remote: Total 59915 (delta 12297), reused 18212 (delta 12202), pack-reused 41604
Receiving objects: 100% (59915/59915), 96.25 MiB | 8.47 MiB/s, done.
Resolving deltas: 100% (40120/40120), done.


┌──(kali㉿kali)-[~/hitb]
└─$ cat linux_kernel_cves/data/kernel_cves.json | jq -r  '[.[] | select(.last_affected_version ≠ null) | select((.last_affected_version | startswith("5.")) or (.last_affected_version | sta
rtswith("6."))) | select(.nvd_text ≠ null and (.nvd_text | contains("heap")) and (.nvd_text | contains("overflow")))] | length'

20


┌──(kali㉿kali)-[~/hitb]
└─$ cat linux_kernel_cves/data/kernel_cves.json | jq -r  '[.[] | select(.last_affected_version ≠ null) | select((.last_affected_version | startswith("5.")) or (.last_affected_version | sta
rtswith("6."))) | select(.nvd_text ≠ null and (.nvd_text | contains("stack")) and (.nvd_text | contains("overflow")))] | length'

6


┌──(kali㉿kali)-[~/hitb]
└─$ cat linux_kernel_cves/data/kernel_cves.json | jq -r  '[.[] | select(.last_affected_version ≠ null) | select((.last_affected_version | startswith("5.")) or (.last_affected_version | sta
rtswith("6."))) | select(.nvd_text ≠ null and (.nvd_text | contains("use-after-free")))] | length'

147


┌──(kali㉿kali)-[~/hitb]
└─$ cat linux_kernel_cves/data/kernel_cves.json | jq -r  '[.[] | select(.last_affected_version ≠ null) | select((.last_affected_version | startswith("5.")) or (.last_affected_version | sta
rtswith("6."))) | select(.nvd_text ≠ null and (.nvd_text | contains("race condition")))] | length'

69
```

Vulnerability distribution by component

Linux 3.x and 4.x



Linux 5.x and 6.x



Analysis of recent kernel vulns

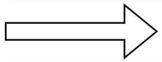
CVE-2022-0435 – A Stack Overflow in TIPC

- TIPC nodes share domain topology information with peers
- Each node stores the most recent domain record from its peers
- The rcv function doesn't check for maximum size of domain members
- The received domain record overflows a 272-bytes buffer on the stack ⟶ **800-byte** overrun

CVE-2022-0435 – A Stack Overflow in TIPC

Pre-overflow:

struct tipc_mon_domain dom_bef	Local variables	**Stack canary**	Callee-saved registers (r12-r15, rbx)	Saved rbp	Saved Return Address

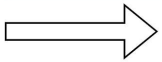


tipc_link_proto_rcv stack frame	tipc_link_rcv stack frame	tipc_rcv stack frame	...	...	...

CVE-2022-0435 – A Stack Overflow in TIPC

Post-overflow:

struct tipc_mon_domain dom_bef	Local variables	**Stack canary**	Callee-saved registers (r12-r15, rbx)	Saved rbp	Saved Return Address

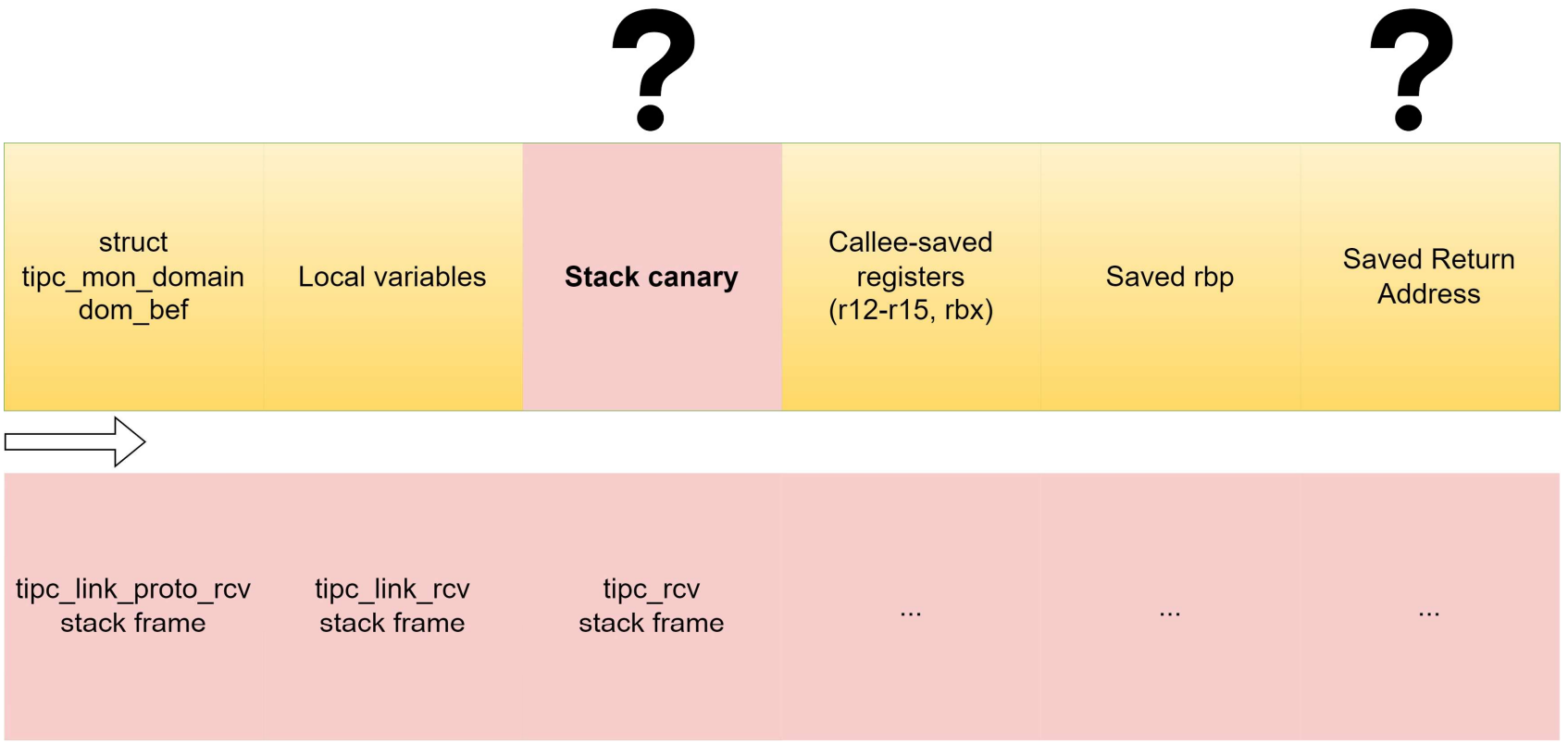


tipc_link_proto_rcv stack frame	tipc_link_rcv stack frame	tipc_rcv stack frame	...	...	...

CVE-2022-0435 – A Stack Overflow in TIPC

- Exploit is remotely triggerable
- Can overwrite the saved RIP
- Powerful exploit primitive
- Cannot corrupt nearby frames, where to pivot?
  - set_memory_x()?
- Exploitation requires disabling CONFIG_STACKPROTECTOR and CONFIG_RANDOMIZE_*

CVE-2022-0435 – A Stack Overflow in TIPC



What else is there?

- KPTI
  - Introduced in 2017
  - Separates page tables between user/kernel mode
- SMEP/SMAP
  - Segregates user mode pages from kernel mode
- FG-KASLR
  - Introduced in 2020, not widely implemented
  - Higher-granularity KASLR
- Per-syscall kernel-stack offset randomization
  - Shifts the kernel stack upon syscalls

The anatomy of a successful Stack Overflow

Bypassing security controls

- KPTI
  - Restore userland page tables: KPTI trampoline
  - User mode helpers (core_*pattern*, *modprobe_path*)
- SMEP/SMAP
  - Never rely on any payload from userland
- FG-KASLR
  - Use symbols from the (.text, .text + 0x400dc6) range
  - Use leaks from the __ksymtab
  - The .data section is still at *.text + k*, user mode helpers!
- Per-syscall kernel-stack offset randomization
  - One-shot exploit
  - **Start from the interrupt context**

An outstanding exploit – CVE-2022-1015

- Nftables subsystem, the newer version of iptables
- OOB r/w primitive on the stack
- Can use the *nft_bitwise* expression to read/write up to 0x40 bytes starting from *&nft_regs + [0x3c0, 0x43c]*
- Input chain will end up in the interrupt context, the output one in the syscall context
- Infoleak from syscall context, memory corruption from the interrupt context

The infoleak

Read OOB into the user NFT registers, where they can be accessed later:

```
gef➤  p &regs
$7 = (struct nft_regs *) 0xffffc90000003a60
gef➤  x/12gx 0xffffc90000003a60-0x5
0xffffc90000003a5b:     0xffffffffff888006      0x347e8000000000ff
0xffffc90000003a6b:     0x008105ffff888006      0x0e274f9f49560e00
0xffffc90000003a7b:     0xffff8880063d85f0      0xffff8880063d8408
0xffffc90000003a8b:     0xffffc90000003a60      0xffff8880063d86d0
0xffffc90000003a9b:     0xffffc90000003c90      0xffffc90000003c60
0xffffc90000003aab:     0xffffffff81c2cfa1      0x0000000100db5fd8
```

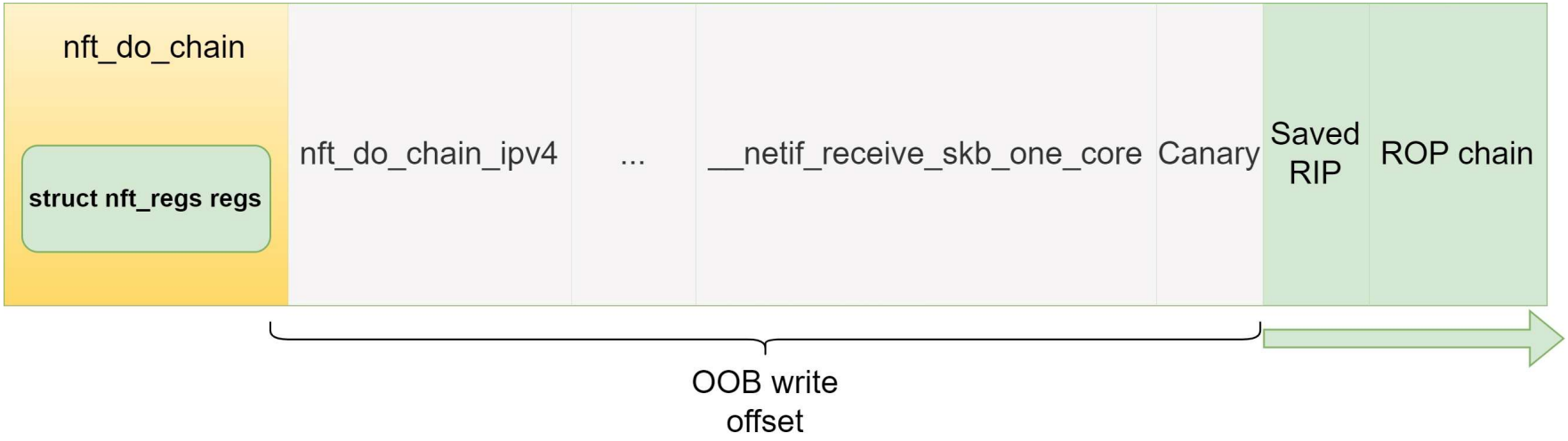
Arbitrary code execution

- **Goal**: Find a return address on a stack frame we can OOB write to
- **Constraint**: leave any stack canary untouched

- Switch between input and output hooks, UDP/TCP/IP packets to find a suitable stack frame

Arbitrary code execution

Stack frames from the irq stack, sending a UDP datagram:



A different approach – CVE-2023-0179

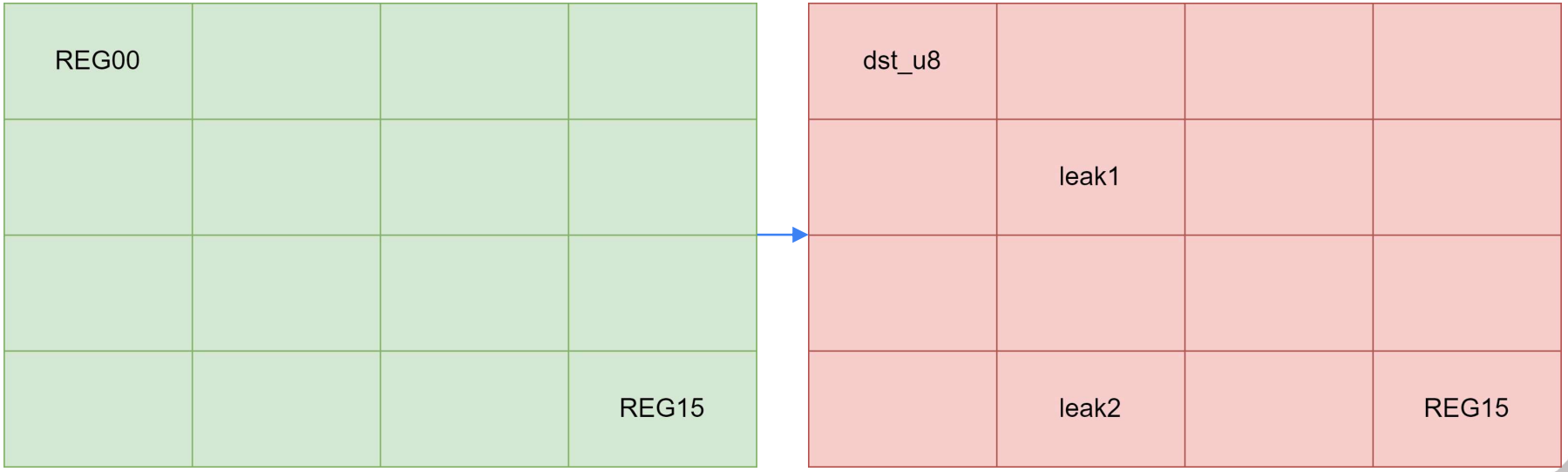
```
1  /* add vlan header into the user buffer for if tag was removed by offloads */
2  static bool
3  nft_payload_copy_vlan(u32 *d, const struct sk_buff *skb, u8 offset, u8 len)
4  {
5      int mac_off = skb_mac_header(skb) - skb->data;
6      u8 *vlanh, *dst_u8 = (u8 *) d;
7      struct vlan_ethhdr veth;
8      u8 vlan_hlen = 0;
9      if ((skb->protocol == htons(ETH_P_8021AD) ||
10          skb->protocol == htons(ETH_P_8021Q)) &&
11         offset >= VLAN_ETH_HLEN && offset < VLAN_ETH_HLEN + VLAN_HLEN)
12         vlan_hlen += VLAN_HLEN;
13     vlanh = (u8 *) &veth;
14     if (offset < VLAN_ETH_HLEN + vlan_hlen) {
15         u8 ethlen = len;
16         if (vlan_hlen &&
17             skb_copy_bits(skb, mac_off, &veth, VLAN_ETH_HLEN) < 0)
18             return false;
19         else if (!nft_payload_rebuild_vlan_hdr(skb, mac_off, &veth))
20             return false;
21         if (offset + len > VLAN_ETH_HLEN + vlan_hlen)
22             ethlen -= offset + len - VLAN_ETH_HLEN + vlan_hlen;
23         memcpy(dst_u8, vlanh + offset - vlan_hlen, ethlen);
```

A different approach – CVE-2023-0179

- Classical stack overflow in the Nftables component (Netfilter)
- Weak exploit primitive: 251-byte overrun starting from *struct nft_regs regs* in the *nft_do_chain* function
- Not enough for stack smashing

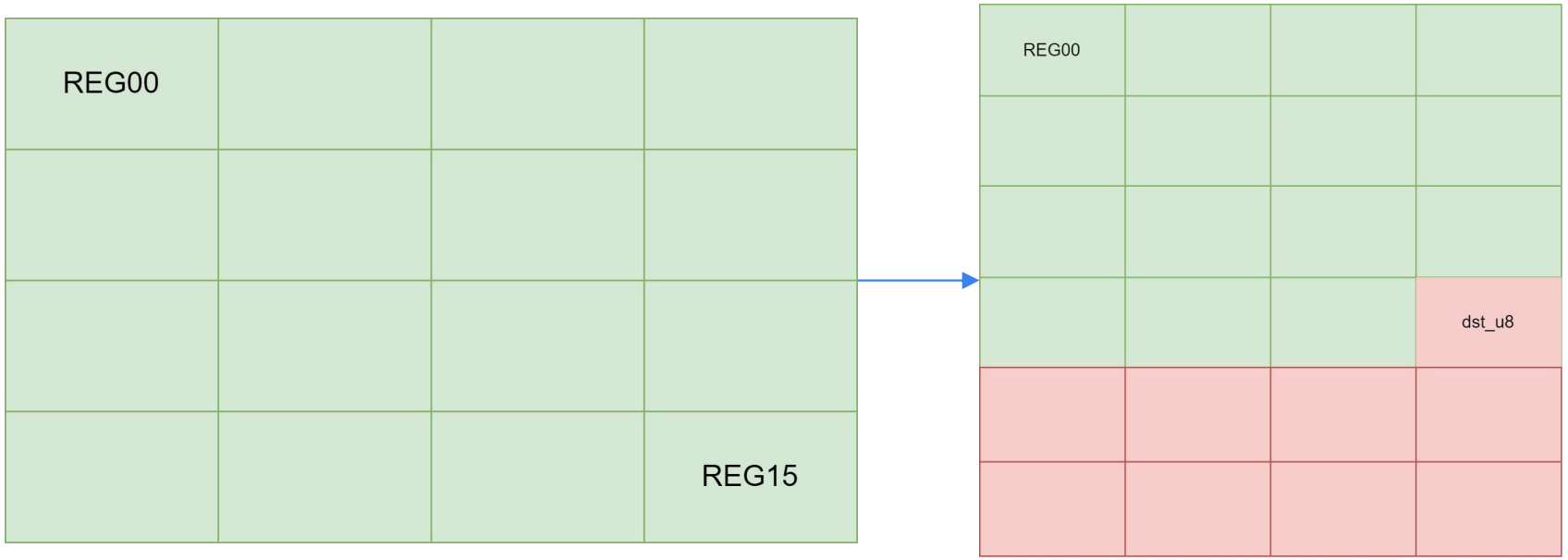
The infoleak

- Use REG00 as destination register for the overflow
- Search for kernel addresses to defeat KASLR



Memory corruption

- Overwriting adjacent memory leads to a protection fault
- A second look revealed a reachable function pointer



Arbitrary code execution

- We control the *expr* pointer which will be eval'd
- Can jump to arbitrary locations

```
1   static void expr_call_ops_eval(const struct nft_expr *expr,
2                                  struct nft_regs *regs,
3                                  struct nft_pktinfo *pkt)
4   {
5   #ifdef CONFIG_RETPOLINE
6       unsigned long e = (unsigned long)expr->ops->eval;
7   #define X(e, fun) \
8       do { if ((e) == (unsigned long)(fun)) \
9           return fun(expr, regs, pkt); } while (0)
10      ...
11  #undef  X
12  #endif /* CONFIG_RETPOLINE */
13      expr->ops->eval(expr, regs, pkt);
14  }
```

The ROP chain: the issues

- Enough space is needed to store our payload
- We do not want to touch userland (no ret2usr)
- NFT registers only offer 64 bytes of storage
  - Insufficient for a full ROP chain (gadgets + pushed regs)
- Controlling the *expr* pointer is space consuming

The ROP chain: the solution

memcpy also includes our payload in the source data, the controlled space can be doubled!

1. Setup the NFT registers with the stack pivot payload (stage 1)
2. Trigger the vuln, causing the payload to move into the jumpstack (an adjacent structure)
3. Refill the NFT registers with the actual ROP payload (stage 2)
4. Redirect execution to stage 1, which will then jump to stage 2
5. User mode helpers!
6. Leave the clobbered functions without panicking

The ROP chain: the solution

Furthermore:

- RANDOMIZE_KSTACK_OFFSET does not apply to softirq
- Stack pivoting is fairly easy with low offsets (function epilogues)
- KASLR leak stays valid until reboot

Leaving the interrupt context

- swapgs_restore_regs_and_return_to_usermode is not available from softirq
- Deadlocks can be ignored since the network interface will be immediately disabled
- The old syscall stack is restored inside *do_softirq()*
- Any function between the last corrupted one and *do_softirq()* can be used as the return target

Leaving the interrupt context

One possible solution is function #5, *nf_hook_slow*

```
gef➤  bt
#0  nft_payload_eval (expr=0xffff888005e769f0, regs=0xffffc90000083950, pkt=0xffffc90000083b80) at net/netfilter/nft_payload.c:124
#1  0xffffffff81c2cfa1 in expr_call_ops_eval (pkt=0xffffc90000083b80, regs=0xffffc90000083950, expr=0xffff888005e769f0) at net/netf
#2  nft_do_chain (pkt=pkt@entry=0xffffc90000083b80, priv=priv@entry=0xffff888005f42a50) at net/netfilter/nf_tables_core.c:264
#3  0xffffffff81c43b14 in nft_do_chain_netdev (priv=0xffff888005f42a50, skb=<optimized out>, state=<optimized out>) at net/netfilte
#4  0xffffffff81c27df8 in nf_hook_entry_hookfn (state=0xffffc90000083c50, skb=0xffff888005f4a200, entry=0xffff88800591cd08) at ./in
#5  nf_hook_slow (skb=skb@entry=0xffff888005f4a200, state=state@entry=0xffffc90000083c50, e=e@entry=0xffff88800591cd00, s=s@entry=0
#6  0xffffffff81b7abf7 in nf_hook_ingress (skb=<optimized out>) at ./include/linux/netfilter_netdev.h:34
#7  nf_ingress (orig_dev=0xffff888005ff0000, ret=<synthetic pointer>, pt_prev=<synthetic pointer>, skb=<optimized out>) at net/core
#8  __netif_receive_skb_core (pskb=pskb@entry=0xffffc90000083cd0, pfmemalloc=pfmemalloc@entry=0x0, ppt_prev=ppt_prev@entry=0xffffc9
#9  0xffffffff81b7b0ef in __netif_receive_skb_one_core (skb=<optimized out>, pfmemalloc=pfmemalloc@entry=0x0) at net/core/dev.c:548
#10 0xffffffff81b7b1a5 in __netif_receive_skb (skb=<optimized out>) at net/core/dev.c:5603
#11 0xffffffff81b7b40a in process_backlog (napi=0xffff888007a335d0, quota=0x40) at net/core/dev.c:5931
#12 0xffffffff81b7c013 in __napi_poll (n=n@entry=0xffff888007a335d0, repoll=repoll@entry=0xffffc90000083daf) at net/core/dev.c:6498
#13 0xffffffff81b7c493 in napi_poll (repoll=0xffffc90000083dc0, n=0xffff888007a335d0) at net/core/dev.c:6565
#14 net_rx_action (h=<optimized out>) at net/core/dev.c:6676
#15 0xffffffff82200135 in __do_softirq () at kernel/softirq.c:574
```

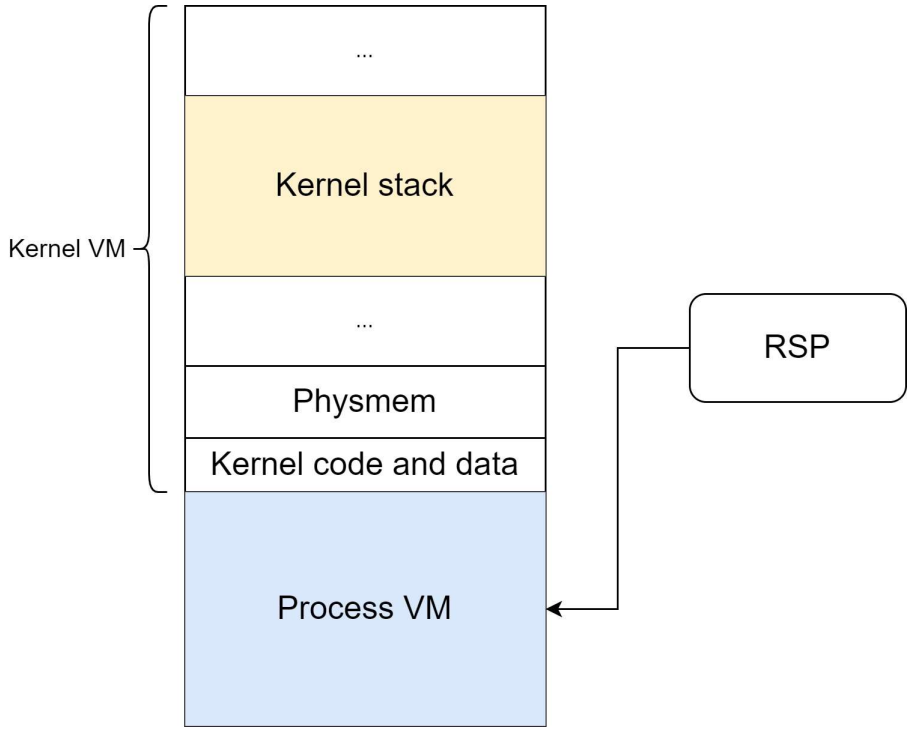
Mitigating the risk inside our kernels

Workarounds

- Reduce the user mode helpers attack surface (CONFIG_STATIC_USERMODEHELPER)
- In-kernel pointer authentication on ARMv8.3+ (since Linux 5.7)
- Per-softirq kernel stack randomization?

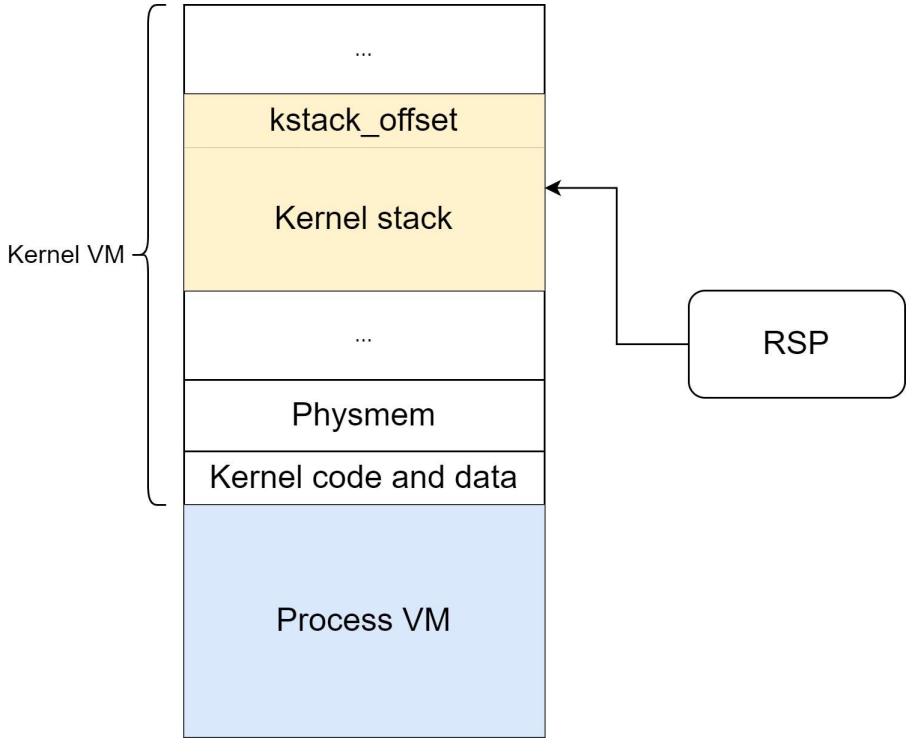
Entering the interrupt context

User context:



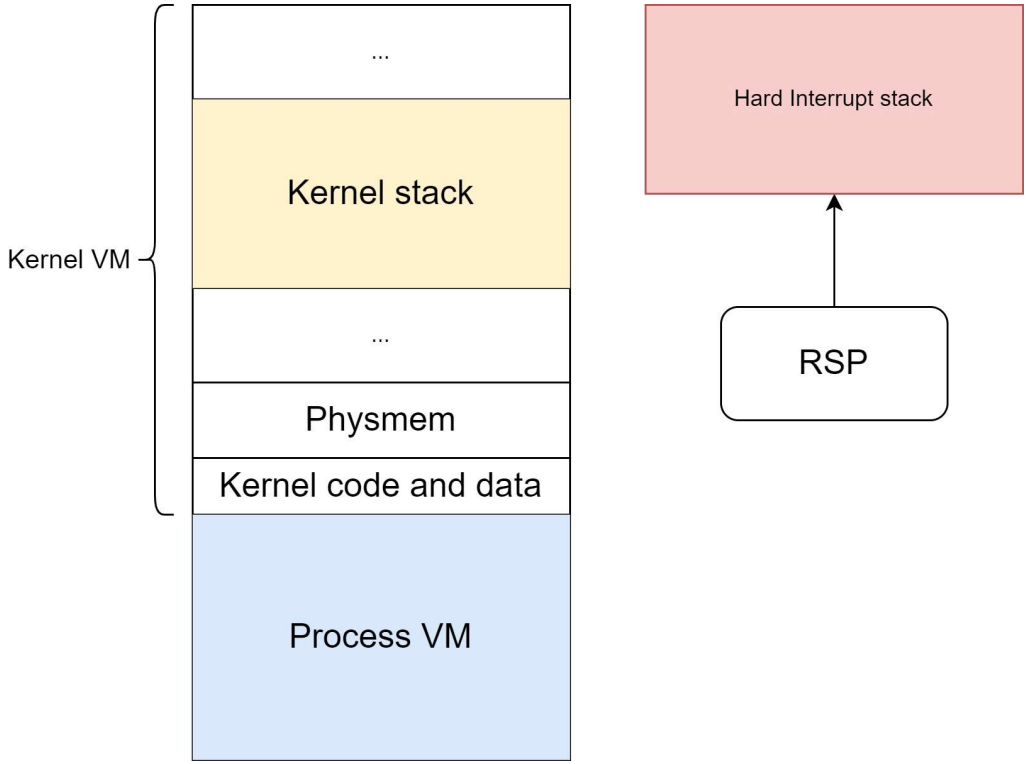
Entering the interrupt context

Syscall context:



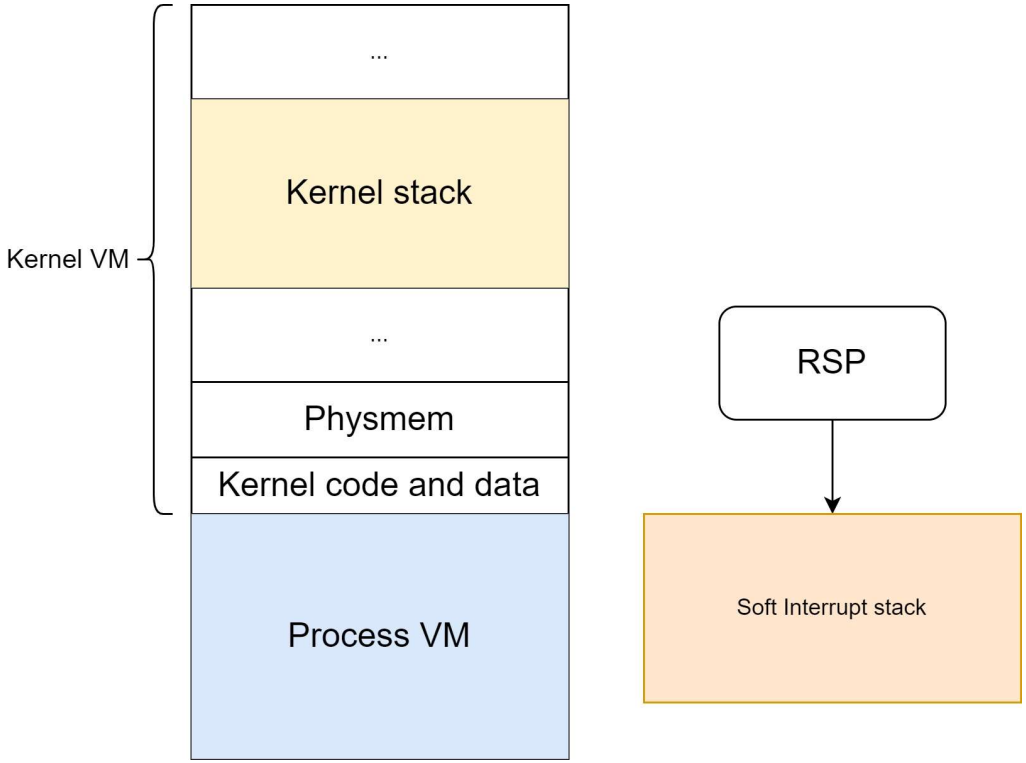
Entering the interrupt context

HardIRQ context:



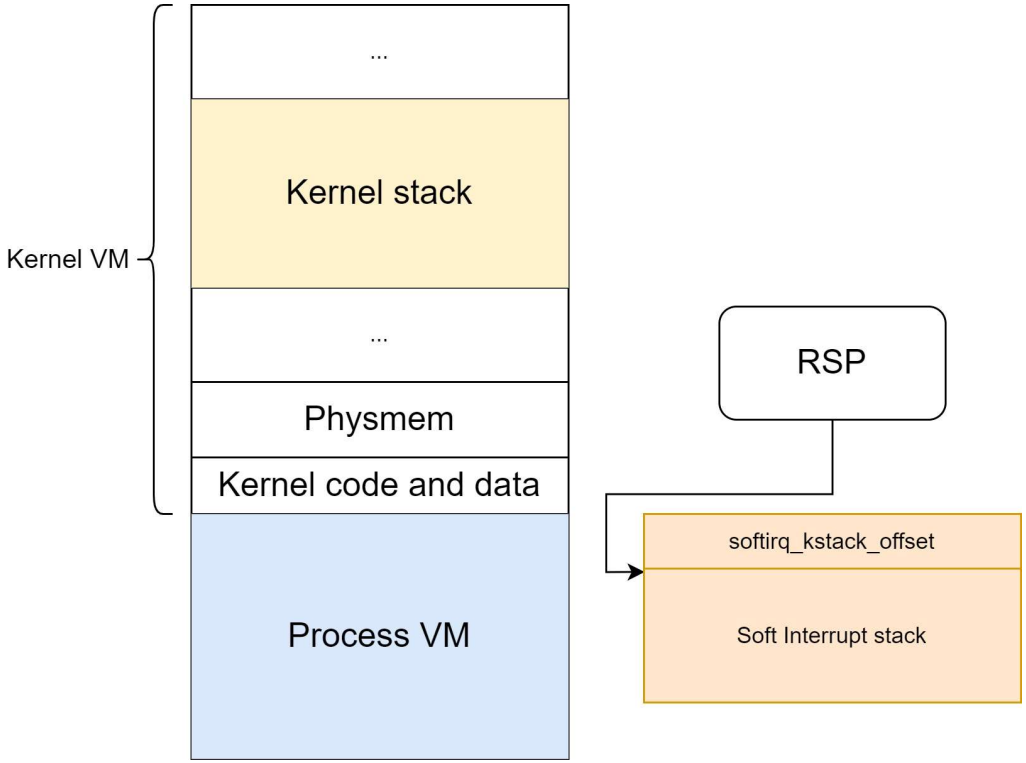
Entering the interrupt context

SoftIRQ context:



Per-softirq kernel stack randomization

SoftIRQ context:



Putting it all together

Stack overflows in the Networking subsystem are still considered powerful since:

- Developers still trust CAP_NET_ADMIN
- RANDOMIZE_KSTACK_OFFSET doesn't work in softirq
  - KASLR leak is reusable
- They easily lead to RCE

#HITB2023AMS https://conference.hitb.org/

Thank you