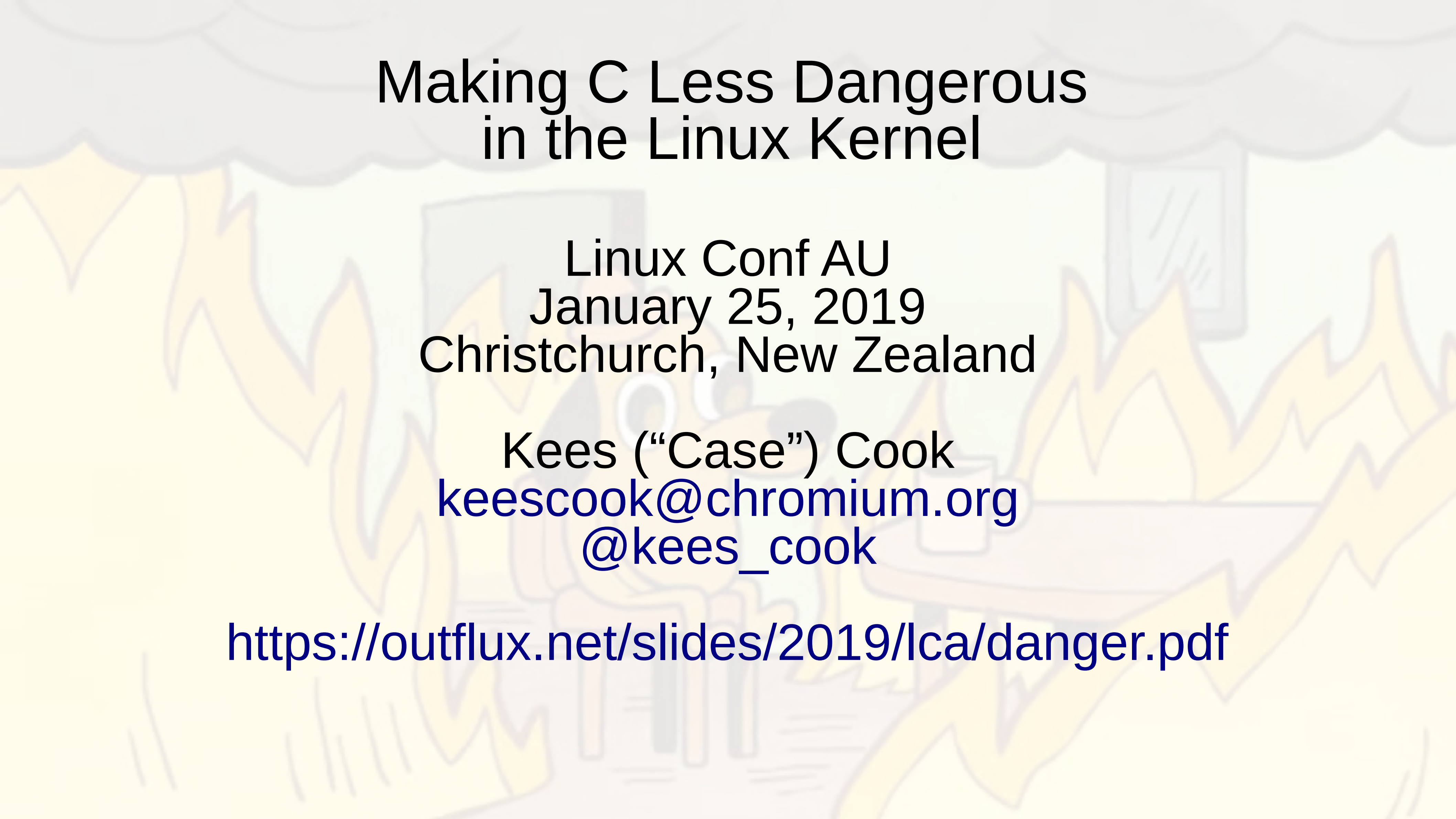
Making C Less Dangerous in the Linux Kernel

Kees Cook | @keescook

LINUX.CONF.AU
21-25 January
2019
Christchurch, NZ

The Linux of Things | #LCA2019 | @linuxconfau



Making C Less Dangerous in the Linux Kernel

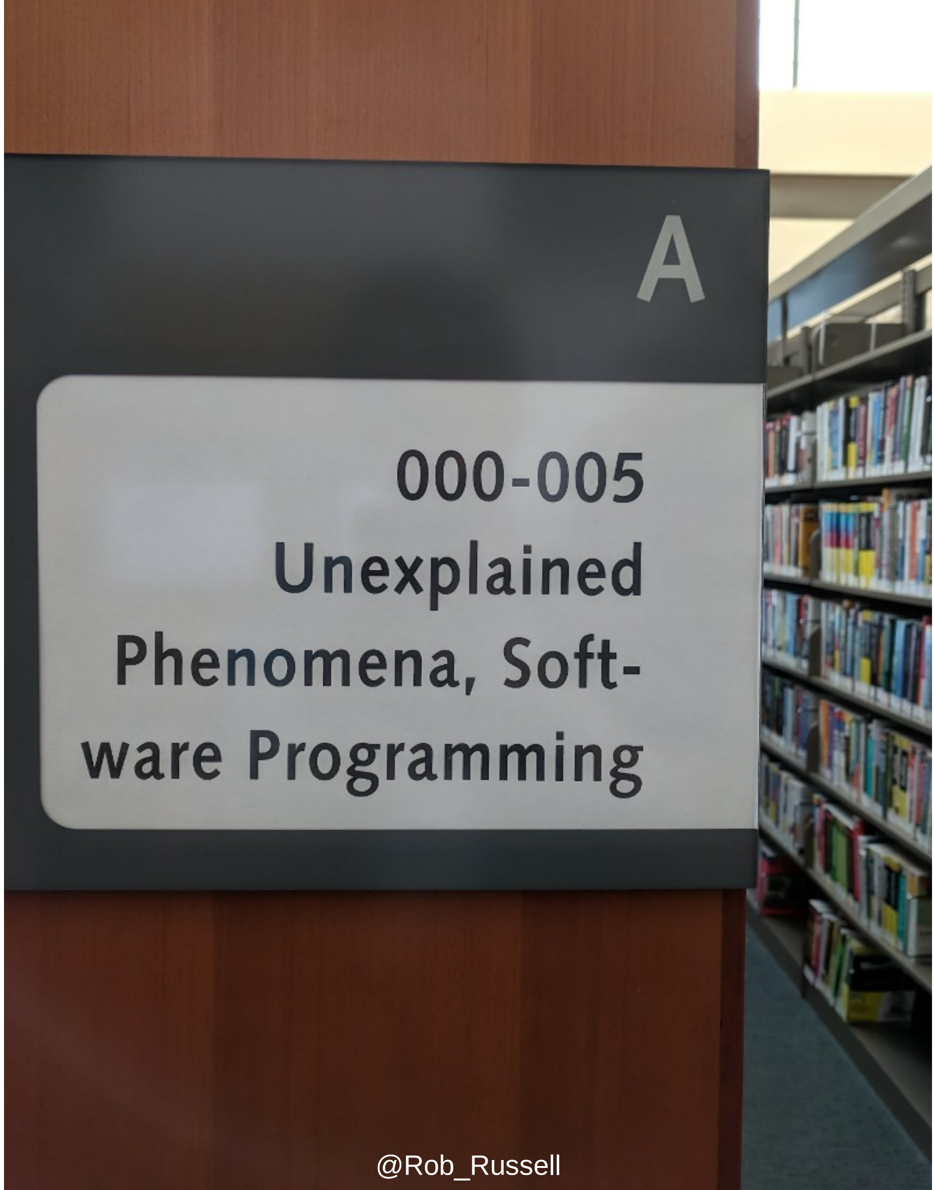
Linux Conf AU
January 25, 2019
Christchurch, New Zealand

Kees ("Case") Cook
keescook@chromium.org
@kees_cook

https://outflux.net/slides/2019/lca/danger.pdf

Agenda

- Background
  - Kernel Self Protection Project
  - C as a fancy assembler
- Towards less dangerous C
  - Variable Length Arrays are bad and slow
  - Explicit switch case fall-through
  - Always-initialized automatic variables
  - Arithmetic overflow detection
  - Hope for bounds checking
  - Control Flow Integrity: forward edges
  - Control Flow Integrity: backward edges
  - Where are we now?
  - How you can help



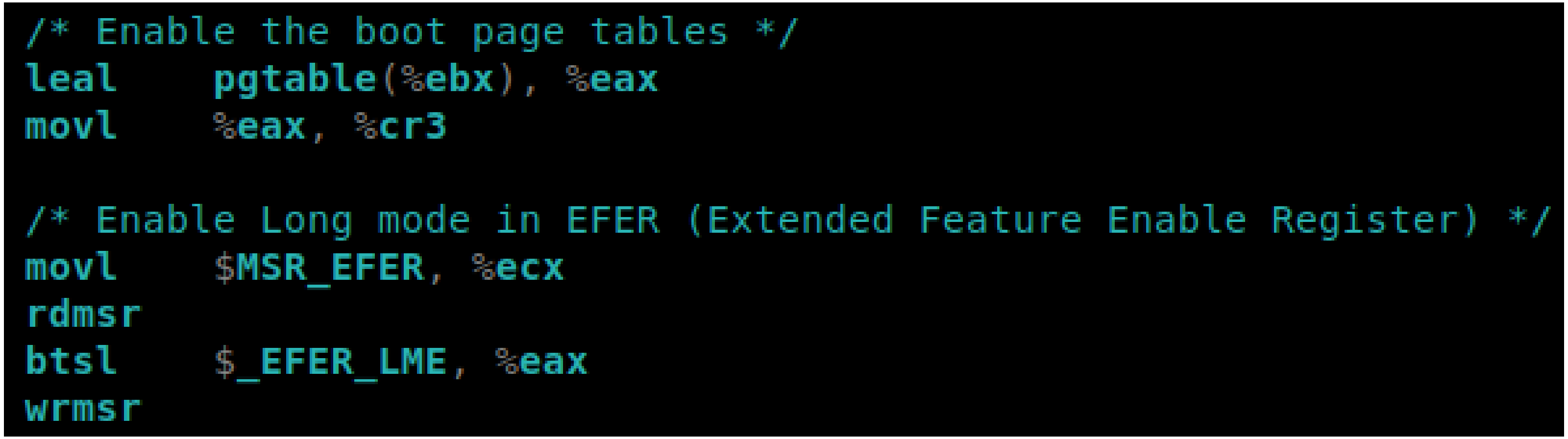
@Rob_Russell

Kernel Self Protection Project

- https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project
- KSPP focuses on the kernel protecting the *kernel* from attack (e.g. refcount overflow) rather than the kernel protecting *userspace* from attack (e.g. execve brute force detection) but any area of related development is welcome
- Currently ~12 organizations and ~10 individuals working on about ~20 technologies
- Slow and steady

C as a fancy assembler: **almost machine code**

- The kernel wants to be as fast and small as possible
- At the core, kernel wants to do very architecture-specific things for memory management, interrupt handling, scheduling, ...
- No C API for setting up page tables, switching to 64-bit mode …



```
/* Enable the boot page tables */
leal    pgtable(%ebx), %eax
movl    %eax, %cr3

/* Enable Long mode in EFER (Extended Feature Enable Register) */
movl    $MSR_EFER, %ecx
rdmsr
btsl    $_EFER_LME, %eax
wrmsr
```

C as a fancy assembler: **undefined behavior**

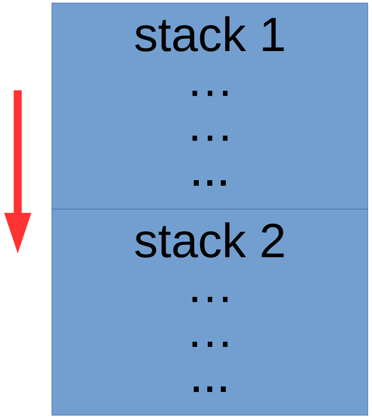
- The C langauge comes with some operational baggage, and weak “standard” libraries
  - What are the contents of “uninitialized” variables?
    - … whatever was in memory from before now!
  - `void` pointers have no type yet we can call typed functions through them?
    - … assembly doesn’t care: everything can be an address to call!
  - Why does `memcpy()` have no “max destination length” argument?
    - … just do what I say; memory areas are all the same!
- “With undefined behavior, anything is possible!”
  - https://raphlinus.github.io/programming/rust/2018/08/17/undefined-behavior.html

Variable Length Arrays (and `alloca()`) are **bad**

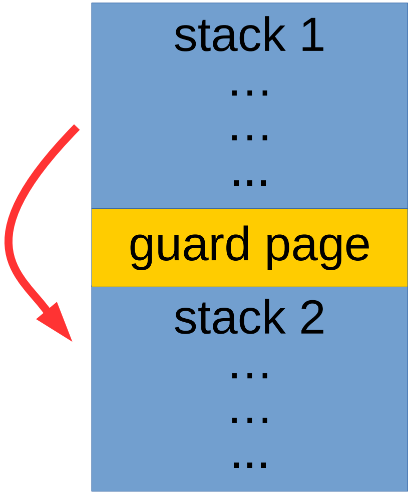
- Exhaust stack, linear overflow: write to things following it
- Jump over guard pages and write to things following it
- Easy to find with compiler flag: `-Wvla`
- But if you must (in userspace) please use gcc's stack probing feature:

```
-fstack-clash-protection
```

```
size = 8192;
...
char buf[size];
...
strcpy(buf, src, size);
```



```
size = 8192;
...
u8 array[size];
...
array[big] = foo;
```



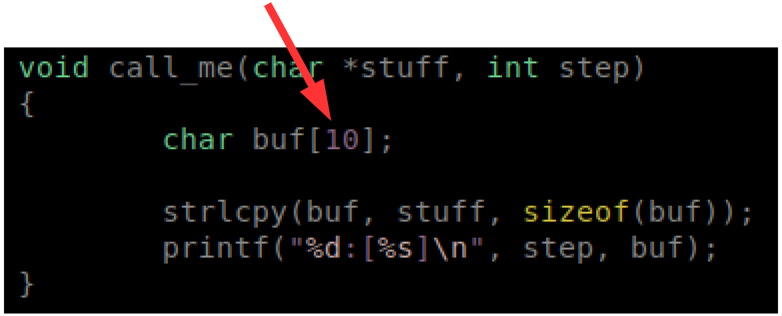
Variable Length Arrays are **slow**

- This seems conceptually sound: more instructions to change stack size, but it seems like it would be hard to measure.
- It is quite measurable … 13% speed up measured during `lib/bch.c` VLA removal:

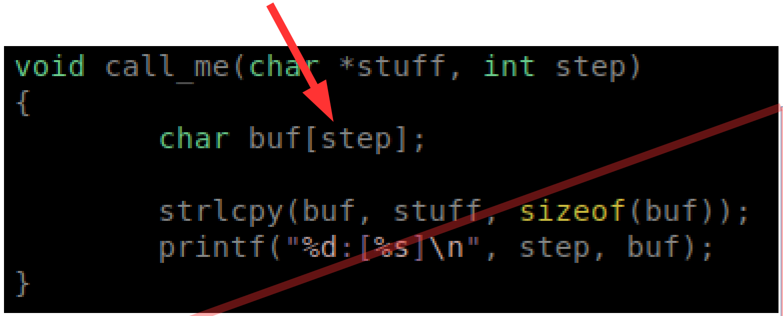
  https://git.kernel.org/linus/02361bc77888 (Ivan Djelic)

```
  Buffer allocation |  Encoding throughput (Mbit/s)
 ---------------------------------------------------
 on-stack, VLA      |   3988
 on-stack, fixed    |   4494
 kmalloc            |   1967
```

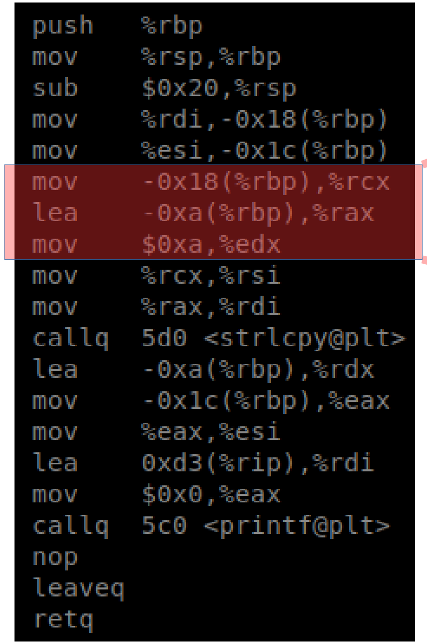
Variable Length Arrays: **stop it**



```c
void call_me(char *stuff, int step)
{
        char buf[10];

        strlcpy(buf, stuff, sizeof(buf));
        printf("%d:[%s]\n", step, buf);
}
```



```c
void call_me(char *stuff, int step)
{
        char buf[step];

        strlcpy(buf, stuff, sizeof(buf));
        printf("%d:[%s]\n", step, buf);
}
```



```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %rdi,-0x18(%rbp)
mov    %esi,-0x1c(%rbp)
mov    -0x18(%rbp),%rcx
lea    -0xa(%rbp),%rax
mov    $0xa,%edx
mov    %rcx,%rsi
mov    %rax,%rdi
callq  5d0 <strlcpy@plt>
lea    -0xa(%rbp),%rdx
mov    -0x1c(%rbp),%eax
mov    %eax,%esi
lea    0xd3(%rip),%rdi
mov    $0x0,%eax
callq  5c0 <printf@plt>
nop
leaveq
retq
```

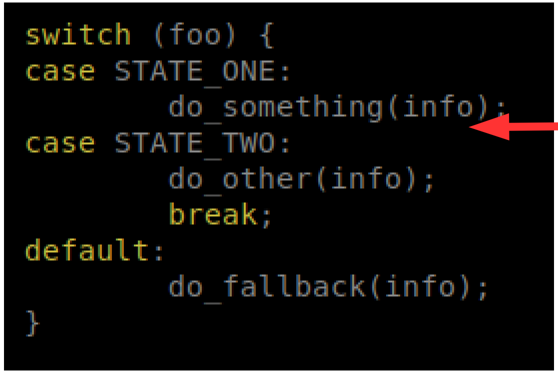
fixed-size array

variable length array

```
push   %rbp
mov    %rsp,%rbp
push   %rbx
sub    $0x28,%rsp
mov    %rdi,-0x28(%rbp)
mov    %esi,-0x2c(%rbp)
mov    %rsp,%rax
mov    %rax,%rbx
mov    -0x2c(%rbp),%ecx
movslq %ecx,%rax
sub    $0x1,%rax
mov    %rax,-0x18(%rbp)
movslq %ecx,%rax
mov    %rax,%r10
mov    $0x0,%r11d
movslq %ecx,%rax
mov    %rax,%r8
mov    $0x0,%r9d
movslq %ecx,%rax
mov    $0x10,%edx
sub    $0x1,%rdx
add    %rdx,%rax
mov    $0x10,%esi
mov    $0x0,%edx
div    %rsi
imul   $0x10,%rax,%rax
sub    %rax,%rsp
mov    %rsp,%rax
add    $0x0,%rax
mov    %rax,-0x20(%rbp)
movslq %ecx,%rdx
mov    -0x20(%rbp),%rax
mov    -0x28(%rbp),%rcx
mov    %rcx,%rsi
mov    %rax,%rdi
callq  5d0 <strlcpy@plt>
mov    -0x20(%rbp),%rdx
mov    -0x2c(%rbp),%eax
```

Switch case fall-through: **did I mean it?**

- CWE-484 “Omitted Break Statement in Switch”
- Semantic weakness in C (“switch” is just assembly test/jump...)
- Commit logs with “missing break statement”: 67



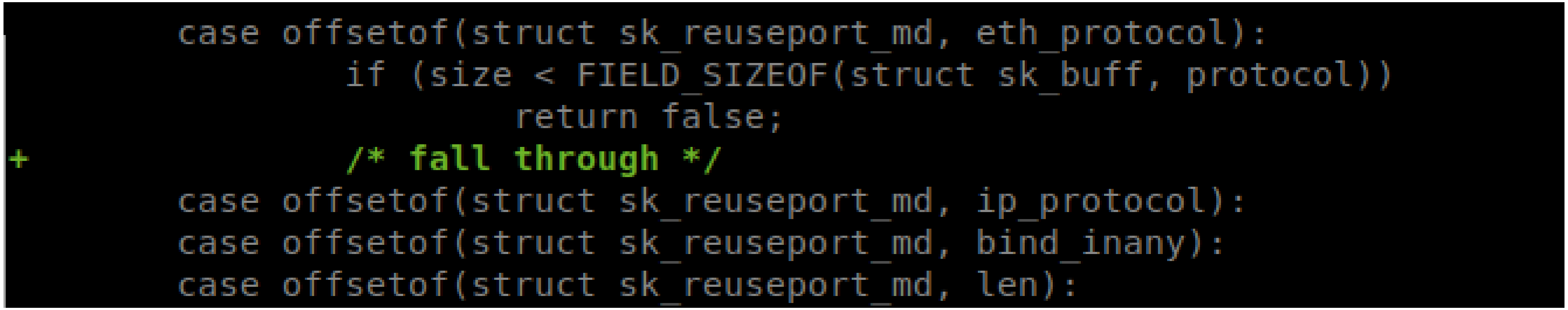
```
switch (foo) {
case STATE_ONE:
        do_something(info);
case STATE_TWO:
        do_other(info);
        break;
default:
        do_fallback(info);
}
```

Did they mean to leave out “`break;`” ??

Switch case fall-through: **new “statement”**

- Use `-Wimplicit-fallthrough` to add a new switch “statement”
  - Actually a comment, but is parsed by compilers now, following the lead of static checkers
- Mark all non-breaks with a “fall through” comment, for example

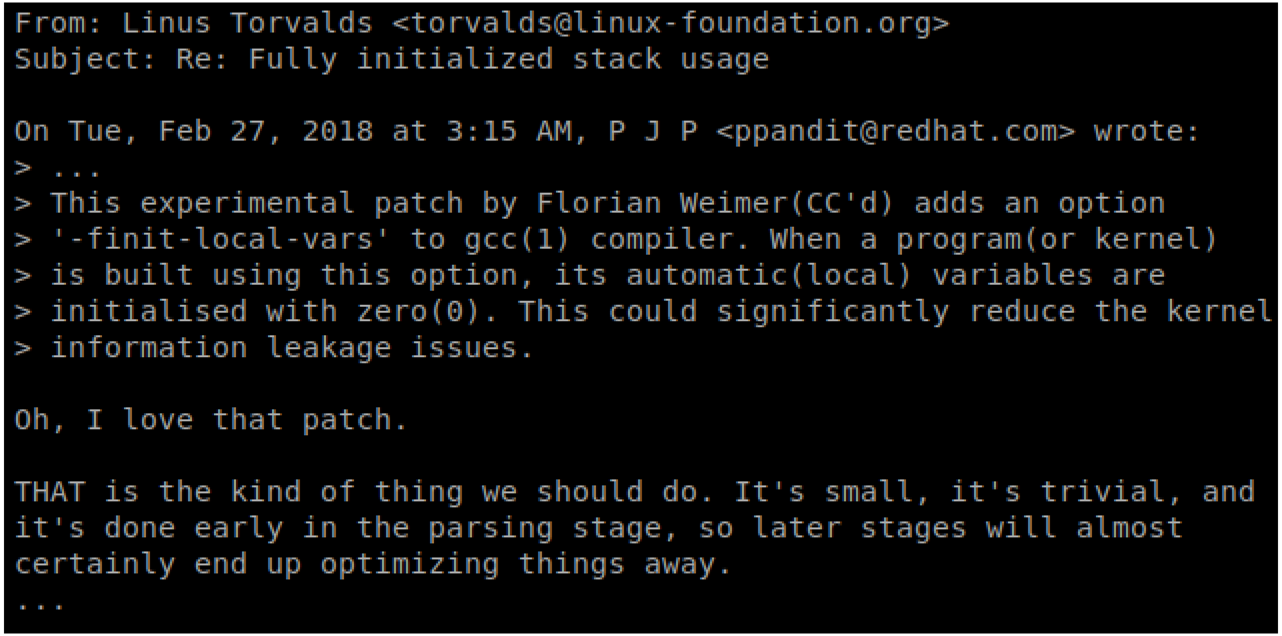
  https://git.kernel.org/linus/4597b62f7a60 (Gustavo A. R. Silva)



```
        case offsetof(struct sk_reuseport_md, eth_protocol):
                if (size < FIELD_SIZEOF(struct sk_buff, protocol))
                        return false;
+               /* fall through */
        case offsetof(struct sk_reuseport_md, ip_protocol):
        case offsetof(struct sk_reuseport_md, bind_inany):
        case offsetof(struct sk_reuseport_md, len):
```

Always-initialized local variables: **just do it**

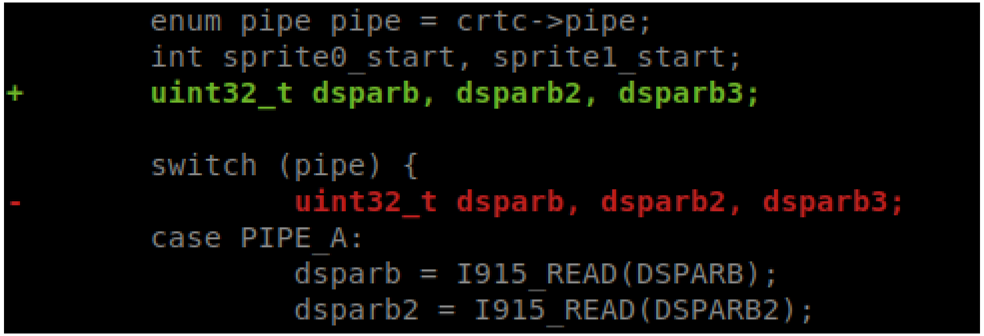
- CWE-200 “Information Exposure”, CWE-457 “Use of Uninitialized Variable”
- gcc `-finit-local-vars` not upstream
- Clang `-fsanitize=init-local` not upstream
- `CONFIG_GCC_PLUGIN_...`
  - `STRUCTLEAK` (for structs with `__user` pointers)
  - `STRUCTLEAK_BYREF` (when passed into funcs)
  - Soon, plugin to mimic `-finit-local-vars` too



```
From: Linus Torvalds <torvalds@linux-foundation.org>
Subject: Re: Fully initialized stack usage

On Tue, Feb 27, 2018 at 3:15 AM, P J P <ppandit@redhat.com> wrote:
> ...
> This experimental patch by Florian Weimer(CC'd) adds an option
> '-finit-local-vars' to gcc(1) compiler. When a program(or kernel)
> is built using this option, its automatic(local) variables are
> initialised with zero(0). This could significantly reduce the kernel
> information leakage issues.

Oh, I love that patch.

THAT is the kind of thing we should do. It's small, it's trivial, and
it's done early in the parsing stage, so later stages will almost
certainly end up optimizing things away.
...
```

Always-initialized local variables: **switch gotcha**

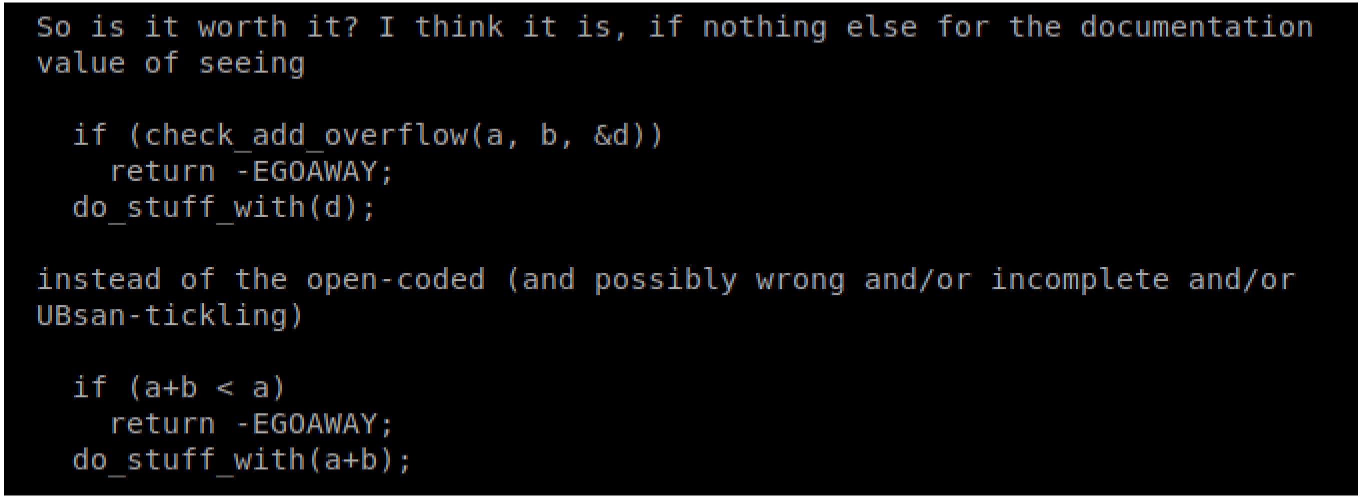
```
warning: statement will never be executed [-Wswitch-unreachable]
```



```
        enum pipe pipe = crtc->pipe;
        int sprite0_start, sprite1_start;
+       uint32_t dsparb, dsparb2, dsparb3;

        switch (pipe) {
-               uint32_t dsparb, dsparb2, dsparb3;
        case PIPE_A:
                dsparb = I915_READ(DSPARB);
                dsparb2 = I915_READ(DSPARB2);
```

Arithmetic overflow detection: **gcc?**

- gcc’s `-fsanitize=signed-integer-overflow` (`CONFIG_UBSAN`)
  - Only signed. Fast: in the noise. Big: warnings grow kernel image by 6% (aborts grow it by 0.1%)
- But we can use explicit single-operation helpers. To quote Rasmus Villemoes:



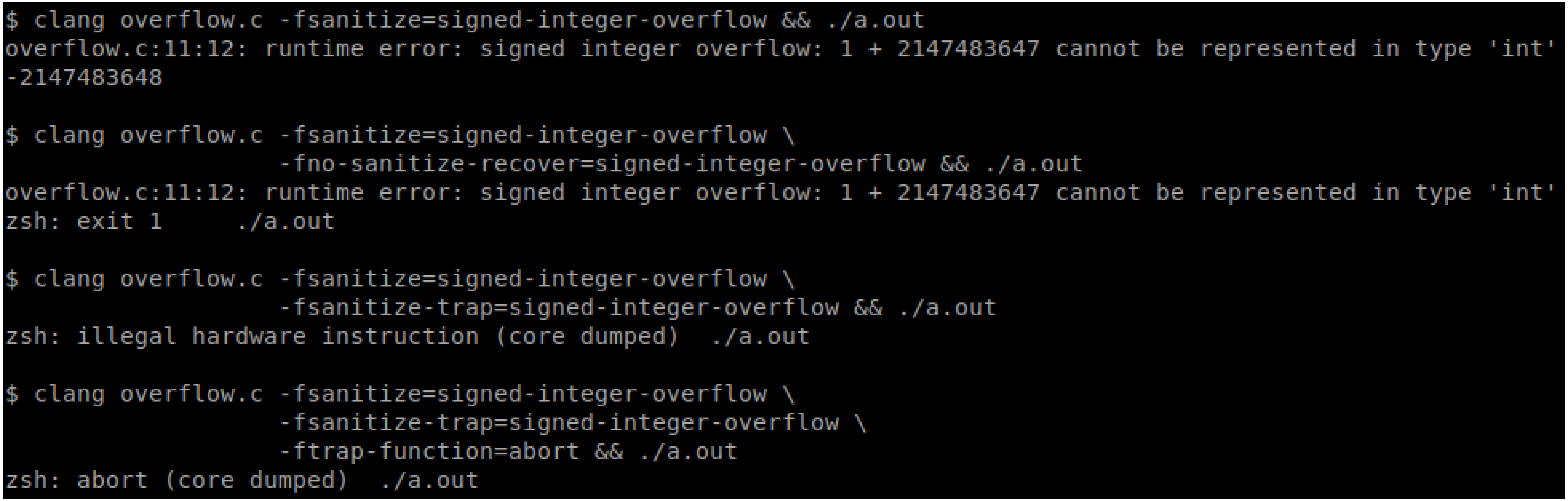
```
So is it worth it? I think it is, if nothing else for the documentation
value of seeing

  if (check_add_overflow(a, b, &d))
    return -EGOAWAY;
  do_stuff_with(d);

instead of the open-coded (and possibly wrong and/or incomplete and/or
UBsan-tickling)

  if (a+b < a)
    return -EGOAWAY;
  do_stuff_with(a+b);
```

Arithmetic overflow detection: **Clang :)**

- Clang can do signed and unsigned instrumentation:

  `-fsanitize=signed-integer-overflow`

  `-fsanitize=unsigned-integer-overflow`



```
$ clang overflow.c -fsanitize=signed-integer-overflow && ./a.out
overflow.c:11:12: runtime error: signed integer overflow: 1 + 2147483647 cannot be represented in type 'int'
-2147483648

$ clang overflow.c -fsanitize=signed-integer-overflow \
                   -fno-sanitize-recover=signed-integer-overflow && ./a.out
overflow.c:11:12: runtime error: signed integer overflow: 1 + 2147483647 cannot be represented in type 'int'
zsh: exit 1     ./a.out

$ clang overflow.c -fsanitize=signed-integer-overflow \
                   -fsanitize-trap=signed-integer-overflow && ./a.out
zsh: illegal hardware instruction (core dumped)  ./a.out

$ clang overflow.c -fsanitize=signed-integer-overflow \
                   -fsanitize-trap=signed-integer-overflow \
                   -ftrap-function=abort && ./a.out
zsh: abort (core dumped)  ./a.out
```

Bounds checking: **explicit checking is slow :(**

- Explicit checks for linear overflows of SLAB objects, stack, etc
  - `copy_{to,from}_user()` checking: <~1% performance hit
  - `strcpy()`-family checking: ~2% performance hit
  - `memcpy()`-family checking: ~1% performance hit
- Can we get better APIs?
  - `strcpy()` is terrible
  - `sprintf()` is bad
  - `memcpy()` is weak

Instead of `strcpy()`: `str`**`s`**`cpy()`

- `strcpy()` no bounds checking on destination nor source!
- `str`**`n`**`cpy()` doesn't always NUL terminate (good for non-C-strings, does NUL pad destination)

```
char dest[4];
strncpy(dest, "ohai!", sizeof(dest)); /* unhelpfully returns dest */
```

`dest: "o", "h", "a",` **`"i"`** … no trailing NUL byte :(

- `str`**`l`**`cpy()` reads source beyond max destination size (returns length of source!)
- `str`**`s`**`cpy()` safest (returns bytes copied, not including NUL, or -E2BIG)

```
ssize_t count = strscpy(dest, "ohai!", sizeof(dest)); /* returns -E2BIG */
dest: "o", "h", "a", NUL
```

  - Does not NUL-pad destination … if desired, add explicit `memset()` (kernel needs a helper for this...)

```
if (count > 0 && count + 1 < sizeof(dest))
    memset(dest + count + 1, 0, sizeof(dest) - count - 1);
```

Instead of `sprintf()`: `sc`**`n`**`printf()`

- `sprintf()` no bounds checking on destination!
- `s`**`n`**`printf()` always NUL-terminates, but returns how much it ***WOULD*** have written :(

```
int count = snprintf(buf, sizeof(buf), fmt…, …);
for (i = 0; i < something; i ++)
  count += snprintf(buf + count, sizeof(buf) - count, fmt…, …);
copy_to_user(user, buf, count);
```

- **`scn`**`printf()` always NUL-terminates, returns count of bytes copied
  - Replace in above code!

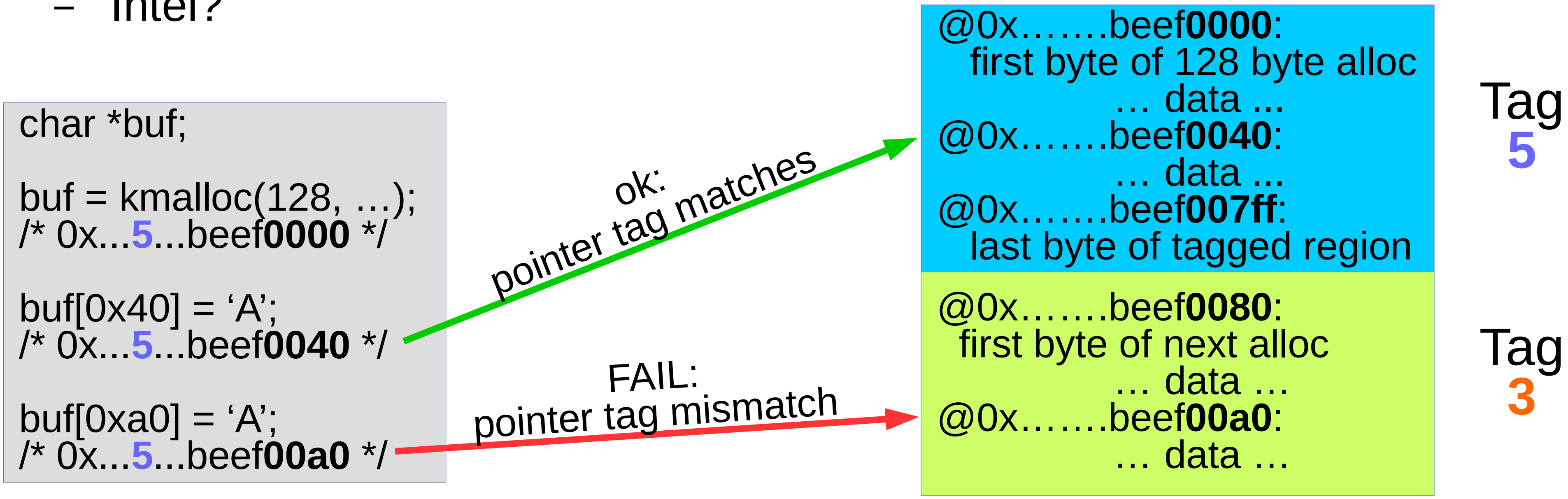
Instead of `memcpy()`: uhhh … be … careful?

- `memcpy()` has no sense of destination size :(

```
uint8_t bytes[128];
size_t wanted, copied = 0;

for (i = 0; i < something && copied < sizeof(bytes); i ++) {
    wanted = ...;
    if (wanted > sizeof(bytes) - copied)
        wanted = sizeof(bytes) - copied;
    memcpy(bytes + copied, wanted, source);
    copied += wanted;
}
```

Bounds checking: **memory tagging :)**

- Hardware memory tagging/coloring is much faster!
  - SPARC Application Data Integrity (ADI)
  - ARMv8.5 Memory Tagging Extension (MTE)
  - Intel?



```
char *buf;

buf = kmalloc(128, ...);
/* 0x...5...beef0000 */

buf[0x40] = 'A';
/* 0x...5...beef0040 */

buf[0xa0] = 'A';
/* 0x...5...beef00a0 */
```

ok:
pointer tag matches

FAIL:
pointer tag mismatch

@0x…….beef**0000**:
first byte of 128 byte alloc
… data ...
@0x…….beef**0040**:
… data ...
@0x…….beef**007ff**:
last byte of tagged region

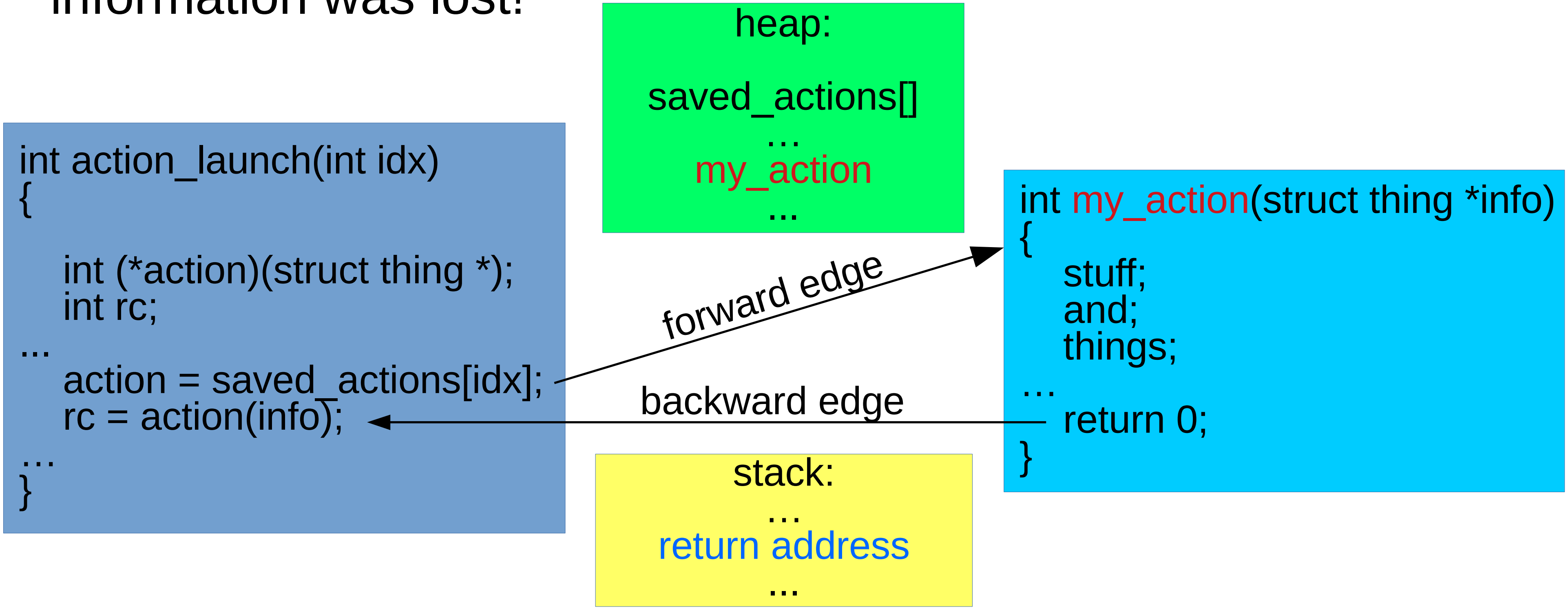
Tag
5

@0x…….beef**0080**:
first byte of next alloc
… data …
@0x…….beef**00a0**:
… data …

Tag
3

Control Flow Integrity: **indirect calls**

- With memory W^X, gaining execution control needs to change function pointers saved in heap or stack, where all type information was lost!



heap:

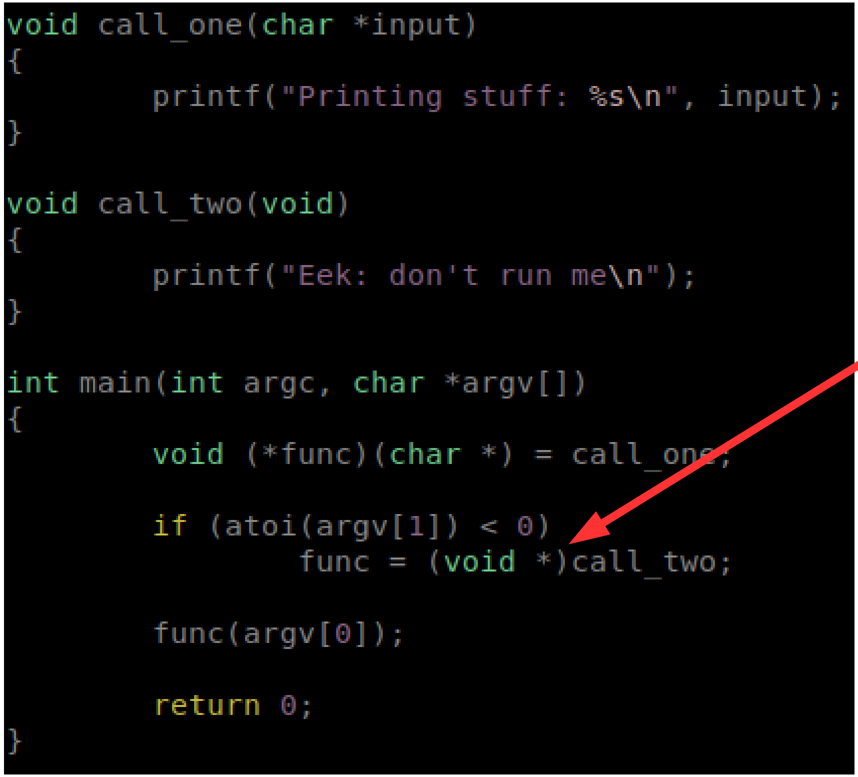
saved_actions[]
…
my_action
…

```
int action_launch(int idx)
{

    int (*action)(struct thing *);
    int rc;
...
    action = saved_actions[idx];
    rc = action(info);
…
}
```

```
int my_action(struct thing *info)
{
    stuff;
    and;
    things;
…
    return 0;
}
```

stack:
…
return address
…

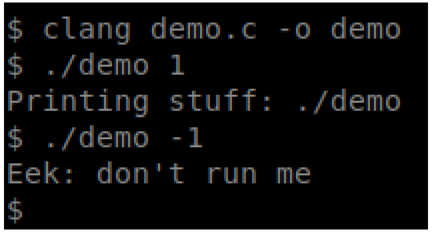
CFI, forward edges: **just call pointers :(**



```
void call_one(char *input)
{
        printf("Printing stuff: %s\n", input);
}

void call_two(void)
{
        printf("Eek: don't run me\n");
}

int main(int argc, char *argv[])
{
        void (*func)(char *) = call_one;

        if (atoi(argv[1]) < 0)
                func = (void *)call_two;

        func(argv[0]);

        return 0;
}
```

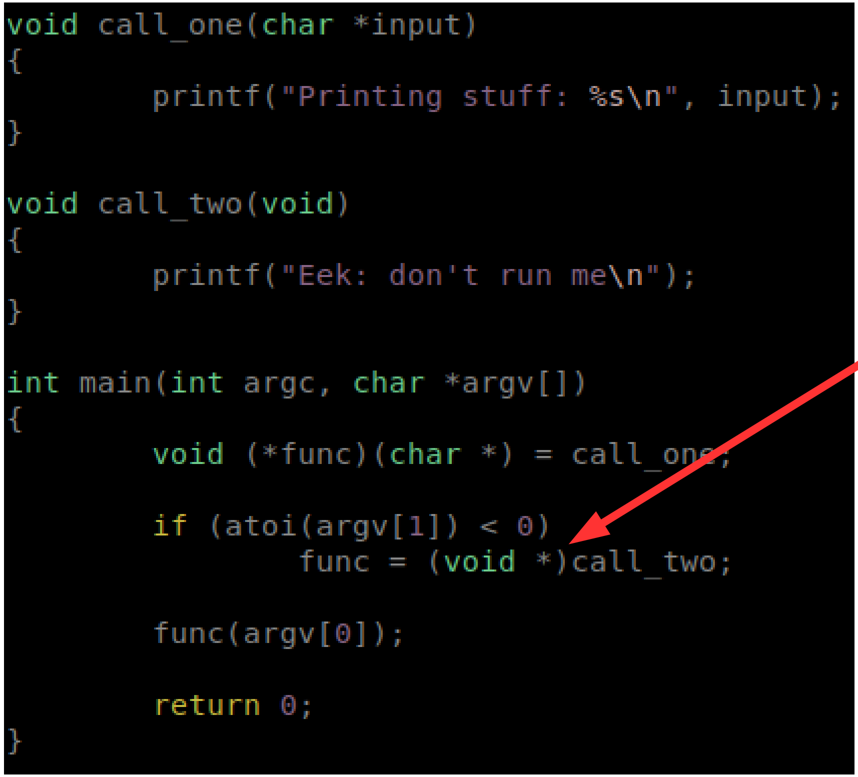
Ignore function prototype …

Normally just a call to a memory address:



```
$ clang demo.c -o demo
$ ./demo 1
Printing stuff: ./demo
$ ./demo -1
Eek: don't run me
$
```

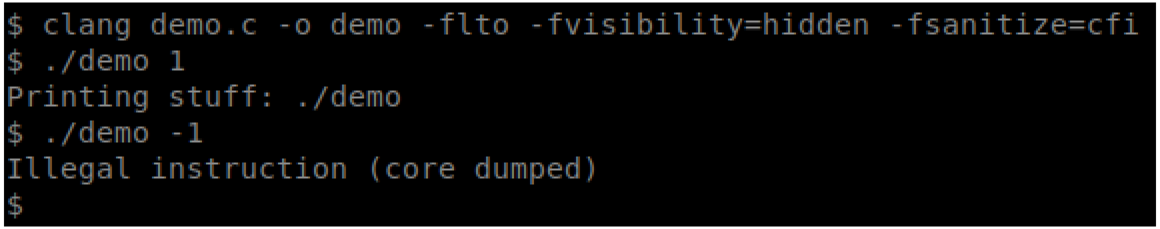
CFI, forward edges: **enforce prototype :)**



```
void call_one(char *input)
{
        printf("Printing stuff: %s\n", input);
}

void call_two(void)
{
        printf("Eek: don't run me\n");
}

int main(int argc, char *argv[])
{
        void (*func)(char *) = call_one;

        if (atoi(argv[1]) < 0)
                func = (void *)call_two;

        func(argv[0]);

        return 0;
}
```

Ignore function prototype …

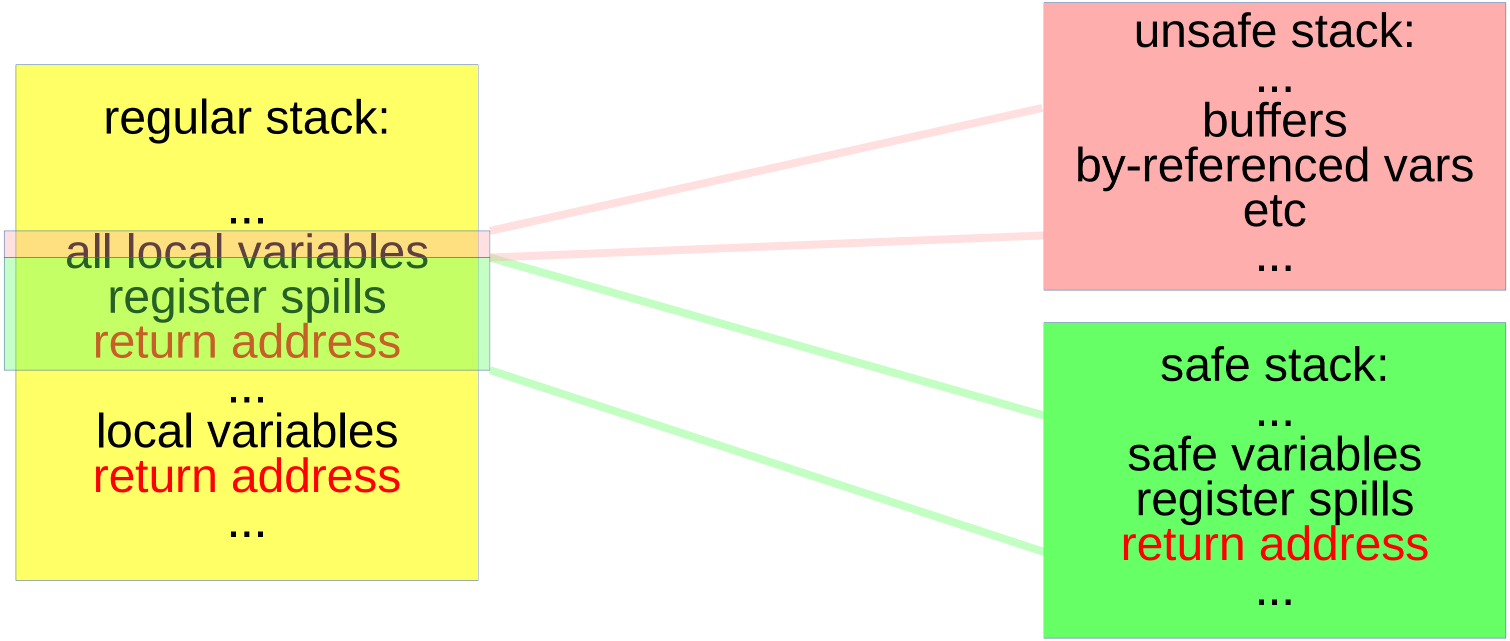
Clang `-fsanitize=cfi` will check at runtime:



```
$ clang demo.c -o demo -flto -fvisibility=hidden -fsanitize=cfi
$ ./demo 1
Printing stuff: ./demo
$ ./demo -1
Illegal instruction (core dumped)
$
```

CFI, backward edges: **two stacks**

- Clang’s Safe Stack
  - Clang: `-fsanitize=safe-stack`



regular stack:

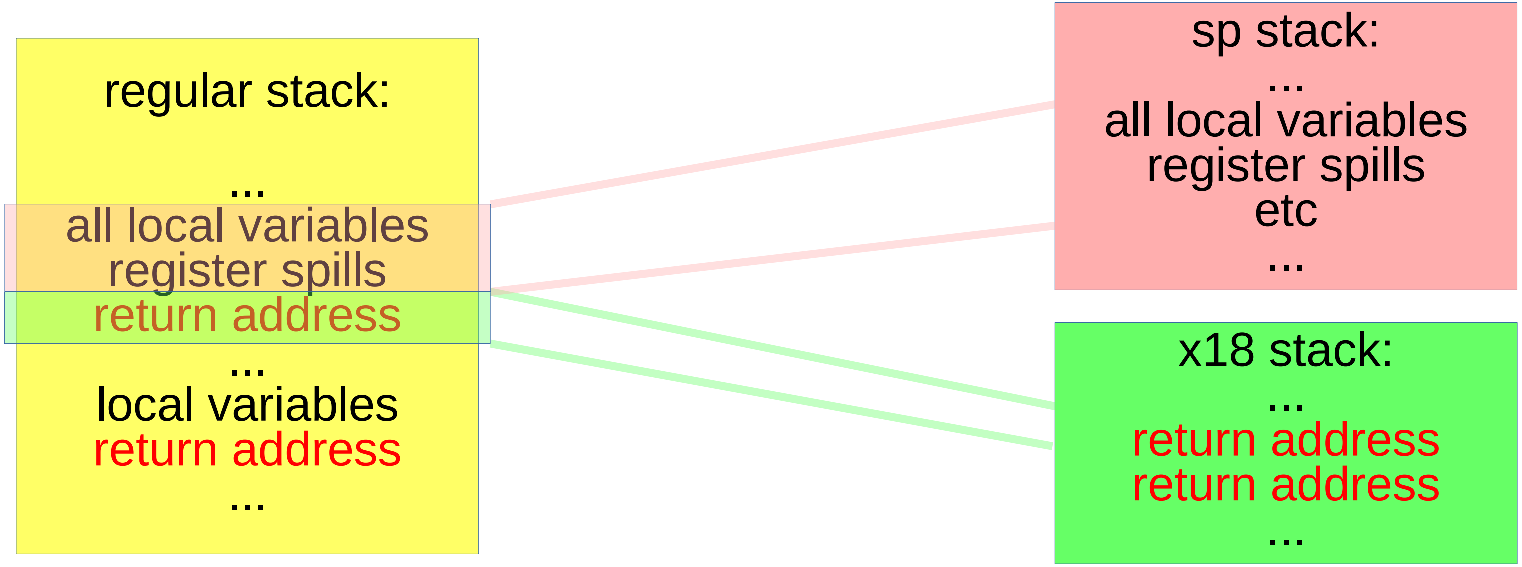
...
all local variables
register spills
return address
...
local variables
return address
...

unsafe stack:
...
buffers
by-referenced vars
etc
...

safe stack:
...
safe variables
register spills
return address
...

CFI, backward edges: **shadow call stack**

- Clang's Shadow Call Stack
  - Clang: `-fsanitize=shadow-call-stack`
  - Results in two stack registers: `sp` and unspilled `x18`



regular stack:

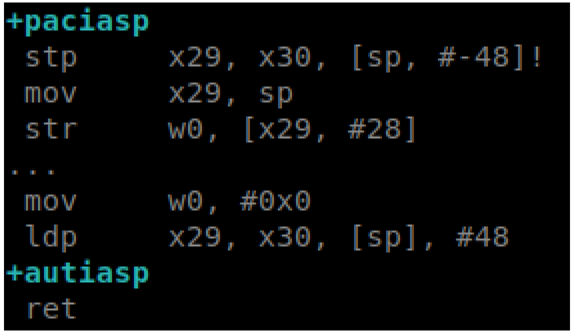
...
all local variables
register spills
return address
...
local variables
return address
...

sp stack:
...
all local variables
register spills
etc
...

x18 stack:
...
return address
return address
...

CFI, backward edges: **hardware support**

- Intel CET: hardware-based read-only shadow call stack
  - Implicit use of otherwise read-only shadow stack during `call` and `ret` instructions
- ARM v8.3a Pointer Authentication (“signed return address”)
  - New instructions: `paciasp` and `autiasp`
  - Clang and gcc: `-msign-return-address`



```
+paciasp
 stp     x29, x30, [sp, #-48]!
 mov     x29, sp
 str     w0, [x29, #28]
...
 mov     w0, #0x0
 ldp     x29, x30, [sp], #48
+autiasp
 ret
```

Where is the Linux kernel now?

- Variable Length Arrays
  - Finally eradicated from kernel since v4.20 (Dec 2018)!
- Explicit switch case fall-through
  - Steady progress on full markings (232 of 2311 remain)
- Always-initialized automatic variables
  - Various partial options, needs more compiler work
- Arithmetic overflow detection
  - Memory allocations now doing explicit overflow detection
  - Needs better kernel support for Clang and gcc
- Bounds checking
  - Crying about performance hits
  - Waiting (im)patiently for hardware support
- Control Flow Integrity: forward edges
  - Need Clang LTO support in kernel, but works on Android
- Control Flow Integrity: backward edges
  - Shadow Call Stack works on Android
  - Waiting (im)patiently for hardware support

https://www.flickr.com/photos/wonderlane/5270711224

Challenges in Kernel Security Development

Cultural: Conservatism, Responsibility, Sacrifice, Patience

Technical: Complexity, Innovation, Collaboration

Resource: Dedicated Developers, Reviewers, Testers, Backporters

Thoughts?

Kees (“Case”) Cook
keescook@chromium.org
keescook@google.com
kees@outflux.net

https://outflux.net/slides/2019/lca/danger.pdf

http://www.openwall.com/lists/kernel-hardening/
http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project