How I started chasing speculative type confusion bugs in the kernel and ended up with ‘real’ ones

Jakob Koschel
PhD student @ Vrije Universiteit Amsterdam
LWN: "The Linux kernel could upgrade from C89 to C11 as early as the 5.18 release"

"Alright, just gonna to patch a few of these Spectre gadgets that we found"
We’ll start with some **Spectre** background

followed by an **interesting case study**

revealing more ‘real’ bugs in the **list iterators**
Speculative Execution & Branch Predictor

char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
i = 0
char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;

// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}

i = 0
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i ++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

Cache

```
i = 0
```
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i ++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

```c
i = 3
```
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

Cache

```
i = 4
```
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;

// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i ++) {
    if (msg[i] != '\0') {
        count += 1;
    }
    else {
        break;
    }
}
i = 6
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
i = 63
```
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

```
i = 64
```
Speculative Execution & Branch Predictor

```c
char msg[128] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

`i = 64`
But what if the CPU is not right?

Can we fool the **Branch Predictor**?
Misprediction

```c
char msg[129] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;

// calculate length of string
for (int i = 0; i < 129; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```
Misprediction

```c
char msg[129] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 129; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

Cache

```
i = 129
```
Misprediction

```c
char msg[129] = "LPCLPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 129; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
i = 129
```
Misprediction

```c
char msg[129] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 129; i ++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
i = 129
```
A Spectre V1 gadget

```c
x = get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
x = get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

Kernel memory

arr1
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
int get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    y = arr1[x];
z = arr2[y];
}
```

Kernel memory

![Kernel memory diagram with arr1 and arr2 arrays]

33
A Spectre V1 gadget

```c
get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

Kernel memory
A Spectre V1 gadget

```c
= get_user(ptr);
if (x < size) {
    = arr1[x];
    z = arr2[y];
}
```
A Spectre V1 gadget

```c
get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

Kernel memory

arr1
What defenses are deployed in the kernel?
What defenses are deployed in the kernel?

lfsence on copy-from-user:

```c
static bool user_access_begin(const void __user *ptr, size_t len)
{
    if (unlikely(!access_ok(ptr, len)))
        return 0;
    __uaccess_begin_nospec();
    return 1;
}
```
What defenses are deployed in the kernel?
What defenses are deployed in the kernel?

```c
static __always_inline bool do_syscall_x64(struct pt_regs *regs, int nr)
{
    unsigned int unr = nr;

    if (likely(unr < NR_syscalls)) {
        unr = array_index_nospec(unr, NR_syscalls);
        regs->ax = sys_call_table[unr](regs);
        return true;
    }
    return false;
}
```
What defenses are deployed in the kernel?
What defenses are deployed in the kernel?

For the Spectre variant 1, vulnerable kernel code (as determined by code audit or scanning tools) is annotated on a case by case basis to use nospec accessor macros for bounds clipping to avoid any usable disclosure gadgets.

What defenses are deployed in the kernel?

For the Spectre variant 1, vulnerable kernel code (as determined by code audit or scanning tools) is annotated on a case by case basis to use nospec accessor macros for bounds clipping to avoid any usable disclosure gadgets. However, it may not cover all attack vectors for Spectre variant 1.
We can do better.

So Brian Johannesmeyer and I started with a dynamic analysis approach in 2019.
We’re using something called Dynamic Taint Analysis, but what is it?
int main(int argc, char *argv[]) {

    char *prog = malloc(100);
    strcpy(prog, argv[1]);

    execve(prog,
            (char *[]){{prog, 0},
                  environ});
}

Dynamic Taint Analysis
int main(int argc, char *argv[]) {

    char *prog = malloc(100);
    strcpy(prog, argv[1]);

    execve(prog, 
            (char *[]) {prog, 0},
            environ);
}
Dynamic Taint Analysis

```c
int main(int argc, char *argv[]) {
    dfsan_add_label(user, argv[1], strlen(argv[1]));

    char *prog = malloc(100);
    strcpy(prog, argv[1]);

    execve(prog,
           (char *[]) {prog, 0},
           environ);
}
```
int main(int argc, char *argv[]) {
    dfsan_add_label(user, argv[1], strlen(argv[1]));
    char *prog = malloc(100);
    strcpy(prog, argv[1]);

    execve(prog,
           (char *[]){prog, 0},
           environ);
}
Dynamic Taint Analysis

```c
int main(int argc, char *argv[]) {
    dfsan_add_label(user, argv[1], strlen(argv[1]));
    char *prog = malloc(100);
    strcpy(prog, argv[1]);
    execve(prog,
           (char *[]){prog, 0},
           environ);
}
```

Taint Source

Taint Propagation

Taint Sink

Violation detected!
Compiler-based dynamic taint analysis in the kernel?
Compiler-based dynamic taint analysis in the kernel?

We’ve built **KDFSAN** for this project!

https://github.com/vusec/kdfsan-linux/tree/kdfsan-linux-v5.13.7
Our approach:
Our approach:

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```

2. Add an attacker label

```
x = -7   x = 3   x = 100000
```
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}

Our approach:

1. **Fuzz** the syscall interface

2. Add an *attacker* label

---

x = -7     x = 3     x = 100000
Our approach:

1. **Fuzz** the syscall interface

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```

2. Add an attacker label

\[ x = -7 \quad x = 3 \quad x = 100000 \]
Our approach:

1. **Fuzz** the syscall interface

2. Add an *attacker* label

3. Start speculative emulation

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an *attacker* label

3. Start speculative emulation

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an attacker label

3. Start speculative emulation

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}

1. **Fuzz** the syscall interface

2. Add an attacker label

3. Start speculative emulation

---

Our approach:

- **Fuzz** the syscall interface with different values of `x`:
  - `x = -7`
  - `x = 3`
  - `x = 100000`

- **Add an attacker label**

- **Start speculative emulation** with modified values of `x`.
Our approach:

1. **Fuzz** the syscall interface

2. Add an *attacker* label

3. Start speculative emulation

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an attacker label

3. Start speculative emulation

4. Memory error detector identifies unsafe access

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an **attacker** label

3. Start **speculative emulation**

4. **Memory error detector** identifies unsafe access

5. Add a **secret** label

```c
void syscall_handler(int x) {
    ... if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an **attacker** label

3. Start **speculative emulation**

4. Memory error detector identifies unsafe access

5. Add a **secret** label

```c
void syscall_handler(int x) {
    ... if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an **attacker** label

3. Start speculative emulation

4. Memory error detector identifies unsafe access

5. Add a **secret** label

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an *attacker* label

3. Start speculative emulation

4. Memory error detector identifies unsafe access

5. Add a *secret* label

6. Cache interference detector identifies gadget

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our approach:

1. **Fuzz** the syscall interface

2. Add an **attacker** label

3. Start speculative emulation

4. **Memory error detector** identifies unsafe access

5. Add a **secret** label

6. **Cache interference detector** identifies gadget

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
void syscall_handler(int x) {
...
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
}

Our approach:

1. **Fuzz** the syscall interface
2. Add an **attacker** label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a **secret** label
6. **Cache interference detector** identifies gadget
7. **Revert** speculative operations

x = -7   x = 3   x = 100000
Our approach:

1. **Fuzz** the syscall interface

3. Start speculative emulation

5. Add a **secret** label

7. **Revert** speculative operations

2. Add an **attacker** label

4. Memory error detector identifies unsafe access

6. Cache interference detector identifies gadget

```c
void syscall_handler(int x) {
    ...
    if (x < size) {
        y = arr1[x];
        z = arr2[y];
    }
}
```
Our implementation: KASPER
Our implementation: KASPER
Our implementation: KASPER

Linux kernel  KASPER runtime libraries
Our implementation: KASPER

Build the kernel with KASPER’s LLVM passes
Our implementation: KASPER

Build the kernel with KASPER’s LLVM passes

Linux kernel

KASPER runtime libraries

KASPER-instrumented kernel
Our implementation: KASPERS

Build the kernel with KASPERS’s LLVM passes

Fuzz the kernel so that KASPERS reports gadgets at runtime
Our implementation: KASPER

1. **Build the kernel** with KASPER’s LLVM passes
2. **Fuzz the kernel** so that KASPER reports gadgets at runtime

- **Linux kernel**
- **KASPER runtime libraries**
- **KASPER-instrumented kernel**
- **Gadgets statistics**
As a result, Kasper discovered 1,379 previously unknown gadgets in the Linux kernel.
Let’s look at a case study.
Background: the list iterator

```c
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
         !list_entry_is_head(pos, head, member);
         pos = list_next_entry(pos, member))
```
Background: the list iterator

```c
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

Data*  Prev  Next
Case study: list iterator

```c
#define list_for_each_entry(pos, head, member)  
   for (pos = list_first_entry(head,  
      typeof(*pos), member);  
   !list_entry_is_head(pos, head, member);  
   pos = list_next_entry(pos, member))
```
Case study: list iterator

```c
#define list_for_each_entry(pos, head, member)  
  for (pos = list_first_entry(head,  
      typeof(*pos), member);  
    !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```
#define list_for_each_entry(pos, head, member)  
for (pos = list_first_entry(head,  
    typeof(*pos), member);  
  !list_entry_is_head(pos, head, member);  
  pos = list_next_entry(pos, member))
Case study: list iterator

```c
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
         !list_entry_is_head(pos, head, member);
         pos = list_next_entry(pos, member))
```

List head
#define list_for_each_entry(pos, head, member) for (pos = list_first_entry(head, typeof(*pos), member); !list_entry_is_head(pos, head, member); pos = list_next_entry(pos, member))
#define list_for_each_entry(pos, head, member)
for (pos = list_first_entry(head, typeof(*pos), member);
    !list_entry_is_head(pos, head, member);
    pos = list_next_entry(pos, member))
Case study: list iterator

Iteration 1

```c
#define list_for_each_entry(pos, head, member)  
  for (pos = list_first_entry(head,  
      typeof(*pos), member);  
  !list_entry_is_head(pos, head, member);  
  pos = list_next_entry(pos, member))
```
Case study: list iterator

Iteration 1

```
#define list_for_each_entry(pos, head, member)  
for (pos = list_first_entry(head,  
    typeof(*pos), member);  
    !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```
Case study: list iterator

Iteration 2

```c
#define list_for_each_entry(pos, head, member)   
   for (pos = list_first_entry(head,           
       typeof(*pos), member);                  
   !list_entry_is_head(pos, head, member);    
   pos = list_next_entry(pos, member))
```
Case study: list iterator

Iteration 2

```c
#define list_for_each_entry(pos, head, member)  
  for (pos = list_first_entry(head,  
    typeof(*pos), member);  
  !list_entry_is_head(pos, head, member);  
  pos = list_next_entry(pos, member))
```

```
Prev Next
```

```
Data*
```

List head
Case study: list iterator

Iteration 2

#define list_for_each_entry(pos, head, member)
for (pos = list_first_entry(head,
    typeof(*pos), member);
    !list_entry_is_head(pos, head, member);
    pos = list_next_entry(pos, member))
Case study: list iterator

Iteration 2

#define list_for_each_entry(pos, head, member) 
  for (pos = list_first_entry(head, 
    typeof(*pos), member); 
  !list_entry_is_head(pos, head, member); 
  pos = list_next_entry(pos, member))
Case study: list iterator

Iteration 3 (termination)

```c
#define list_for_each_entry(pos, head, member)  
  for (pos = list_first_entry(head,  
      typeof(*pos), member);  
    !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```
#define list_for_each_entry(pos, head, member)  
  for (pos = list_first_entry(head,  
           typeof(*pos), member);  
      !list_entry_is_head(pos, head, member);  
      pos = list_next_entry(pos, member))

Case study: list iterator

Iteration 3 (misprediction)
Case study: list iterator

Iteration 3 (misprediction)

```c
#define list_for_each_entry(pos, head, member) 
for (pos = list_first_entry(head, 
    typeof(*pos), member);
!list_entry_is_head(pos, head, member);
    pos = list_next_entry(pos, member))
```

List head

![Diagram of list iterator with pos pointing to list elements and related function definitions.](image)
Case study: list iterator

Iteration 3 (misprediction)

List head

```c
#define list_for_each_entry(pos, head, member)  
   for (pos = list_first_entry(head,  
       typeof(*pos), member);  
   !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```
Case study: list iterator

Iteration 3 (misprediction)

```c
#define list_for_each_entry(pos, head, member)
for (pos = list_first_entry(head, typeof(*pos), member);
    !list_entry_is_head(pos, head, member);
    pos = list_next_entry(pos, member))
```
Case study: list iterator

Iteration 3 (misprediction)

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
         !list_entry_is_head(pos, head, member);
         pos = list_next_entry(pos, member))
```
Case study: list iterator

Iteration 3 (misprediction)
Case study: list iterator

Iteration 3 (misprediction)
Case study: list iterator

Iteration 3 (misprediction)
Case study: list iterator

Iteration 3 (misprediction)

List head

Side Channel

Load buffer

pos

Secret

Secret

Data

Data

Prev

Next

Prev

Next

Prev

Next
Finally, we implemented a proof-of-concept exploit of an instance of this gadget.

But that’s just the beginning of the story...
The issue **cannot** be **solved** with a simple `array_index_nospec()`
How can we fix this?

```c
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
        ({ bool _cond = !list_entry_is_head(pos, head, member);
            pos = select_nospec(_cond, pos, NULL); _cond; });
        pos = list_next_entry(pos, member))
```
The iterator variable is invalidated at the end of the loop.

But 450+ locations use it after the loop.
But 450+ locations use it after the loop.

$ make coccicheck COCCI=scripts/coccinelle/iterators/use_after_iter.cocci

./arch/arm/mach-mmp/sram.c:54:6-10: ERROR: invalid reference to the index variable of the iterator on line 48
./arch/arm/plat-pxa/ssp.c:54:6-9: ERROR: invalid reference to the index variable of the iterator on line 44
./arch/arm/plat-pxa/ssp.c:78:6-9: ERROR: invalid reference to the index variable of the iterator on line 68
./block/blk-mq.c:4481:11-13: ERROR: invalid reference to the index variable of the iterator on line 4472
./drivers/block/rbd.c:776:16-27: ERROR: invalid reference to the index variable of the iterator on line 766
./drivers/dma/at_xdma.c:1583:13-17: ERROR: invalid reference to the index variable of the iterator on line 1571
...
Turns out some of them are actual bugs!
Let’s look at **architectural** bugs now.

No speculation beyond this point...

```
asm("lfence");
```
struct goku_request *req;

list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}

if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}

goto out;
It looks safe, right?

```c
struct goku_request *req;
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ret = -EINVAL;
goto out;
}
```

---

Linux Plumbers Conference | Dublin, Ireland | Sept. 12-14, 2022
struct goku_request *req;
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}
Does it still look safe?

```c
struct goku_request *req;

list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}

if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}
```

---

Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022
Case study
Case study

Iteration 1

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
```
Case study

Iteration 1

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
```
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
Case study

Iteration 1

```
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
```
Case study

Iteration 2

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
```
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
Case study

After loop

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...}
```
Case study

After loop

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req) {
        break;
    }
    if (&req->req != _req) {
        ...
    }
}
```
Case study

After loop

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
```
Case study

After loop

```c
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ...
}
```
Type Confusion in C
Type Confusion in C

- `container_of()` is performed on the `list_head` which is not contained in a struct
Type Confusion in C

- `container_of()` is performed on the `list_head` which is **not contained** in a struct
- It resembles an **invalid downcast** in object oriented programming
Type Confusion in C

- `container_of()` is performed on the `list_head` which is not contained in a struct.
- It resembles an `invalid downcast` in object oriented programming.
- That's why we call it a `type confusion`.
Quotes from Linus
Quotes from Linus

Make the rule be "you never use the iterator outside the loop".
Quotes from Linus

Make the rule be "you never use the iterator outside the loop".

The whole reason this [...] bug can happen is that we [...] didn’t have C99-style "declare variables in loops".
Quotes from Linus

Make the rule be "you never use the iterator outside the loop".

The whole reason this [...] bug can happen is that we [...] didn’t have C99-style "declare variables in loops".

"we could finally start using variable declarations in for-statements"
The correct way
The correct way

```c
struct goku_request *req = NULL, *iter;

list_for_each_entry(iter, &ep->queue, queue) {
    if (&iter->req == _req) {
        req = iter;
        break;
    }
}

if (!req) {
    ret = -EINVAL;
    goto out;
}
```

Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022
Moving the kernel to modern C

By Jonathan Corbet  
February 24, 2022

Despite its generally fast-moving nature, the kernel project relies on a number of old tools. While critics like to focus on the community's extensive use of email, a possibly more significant anachronism is the use of the 1989 version of the C language standard for kernel code — a standard that was codified before the kernel project even began over 30 years ago. It is looking like that longstanding practice could be coming to an end as soon as the 5.18 kernel, which can be expected in May of this year.

Linked-list concerns

The discussion started with this patch series from Jakob Koschel, who is trying to prevent speculative-execution attacks in the kernel by implementing linked lists on top of the small page structures. The discussion is still ongoing and not all parties are convinced of the need to make this change.
Submitting patches is **fun but very time intensive**.

Around 80 patches have been merged so far.
~300 locations still use the list iterator after the loop!

Patching has to be done one by one.
Without fixing them there is no real benefit of moving to C11.
Treewide changesets is a tricky entry to submitting patches.

Same bugs will need different fixes depending on the maintainer.
Knowing how to split them into pieces is difficult.

Different subsystems have different rules.
Big shoutout to Mike Rapoport for his massive help!
There might be more type confusions in the kernel.

Maybe it’s time for a new scanner...
Any use of `container_of()` can potentially result in a type confusion...

Detecting those is ongoing work.
We’ve started with building a speculative gadget scanner…

… ended up with real type confusion bugs

and caused the kernel to move to a more modern version of C.

Thank you!

Jakob-Koschel

@JakobKoschel

j.koschel AT vu DOT nl