Finding User/Kernel Bugs with Type Inference

Rob Johnson and David Wagner
UC Berkeley

User/Kernel Pointer Bugs

```
int x;
void sys_setint(int *p) {
  memcpy(&x, p, sizeof(x)); // BAD!
}
void sys_getint(int *p) {
  memcpy(p, &x, sizeof(x)); // BAD!
}
```

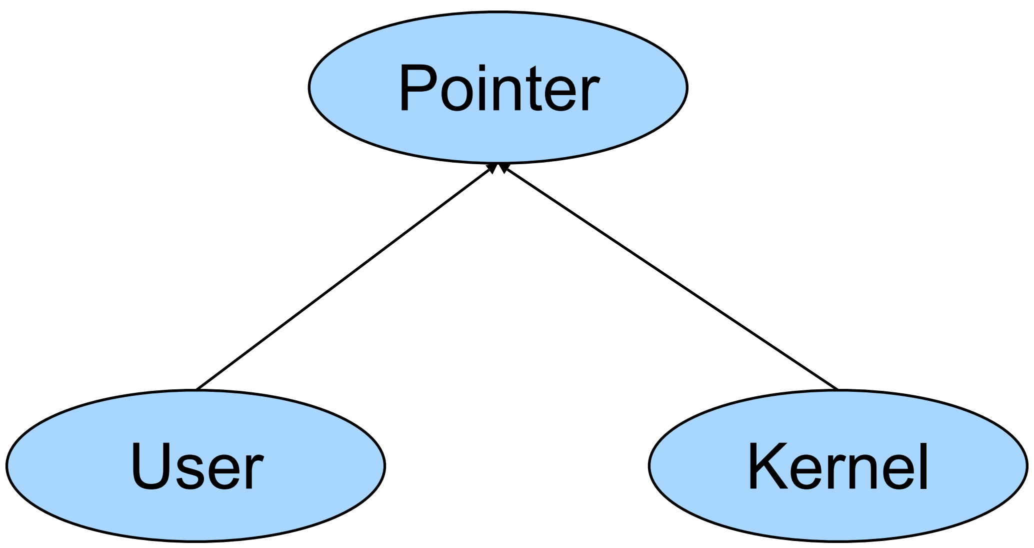
```
getint(buf);
```

- `buf` might point to unmapped memory → page fault
- `buf` might point to kernel region
  - first set then get → can override kernel memory
  - attacker could read arbitrary kernel memory locations

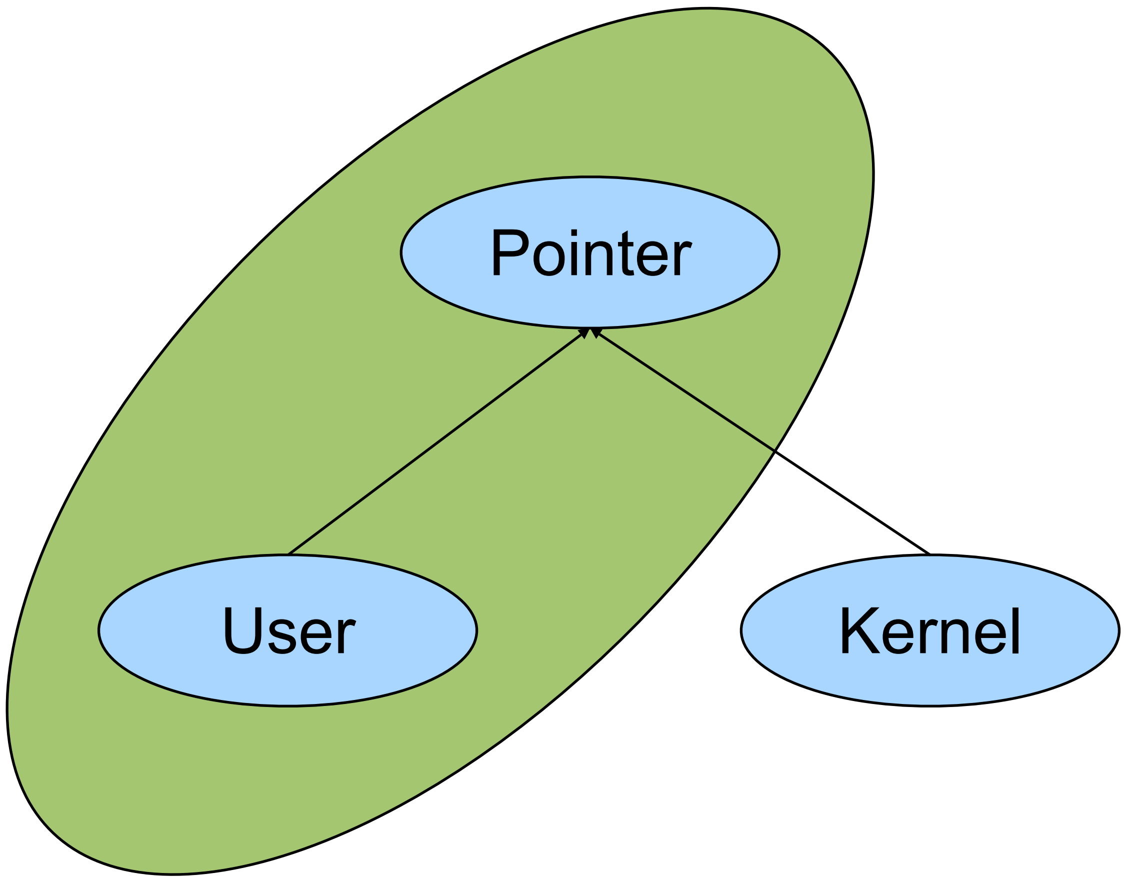
The solution: Different pointer types

- User pointers
  A pointer whose value is under user control and hence untrustworthy
- Kernel pointers
  A pointer variable whose value is under kernel and guaranteed by the kernel to always point into kernel's memory space, and hence is trustworthy
- Relation to ADT
  kernel int is different type than *user int*, so the type checker can check them

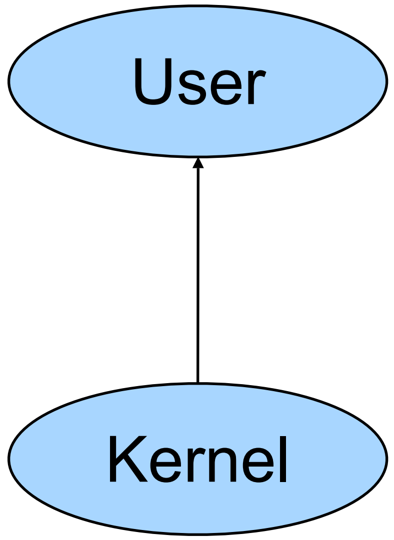
The solution: Different pointer types



The solution: Different pointer types



The solution: Different pointer types



The solution: Different pointer types

```
int copy_from_user(void * kernel to,
                   void * user from, int len);

int memcpy(void * kernel to,
           void * kernel from, int len);

int x;
void sys_setint(int * user p) {
    copy_from_user(&x, p, sizeof(x));
}

void sys_getint(int * user p) {
    memcpy(p, &x, sizeof(x));  // TYPE-CHECK ERROR
}
```

Qualifier inference

- Want to find bugs in Linux kernel which is huge (2.3 Mloc)
- Manually annotating every pointer with a qualifier is infeasible
- Instead: write down qualifiers in a few key places, infer them everywhere else

How inference works

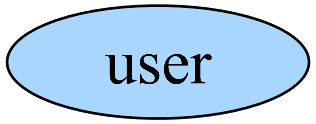
- They use a modified version of CQUAL
  - Uses similar algorithmic idea as Lackwit
- Manually annotate:
  - system calls with *user*
  - dereferences with *kernel*
- Everything in between is inferred.

Qualifier inference example

```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```

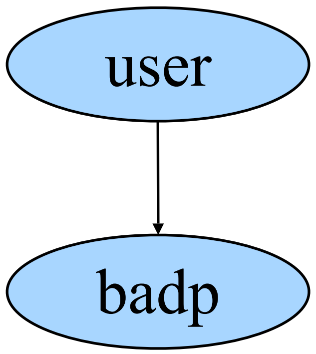
Qualifier inference example

```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```



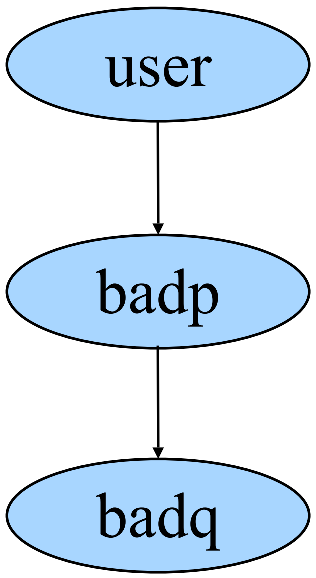
Qualifier inference example

```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```



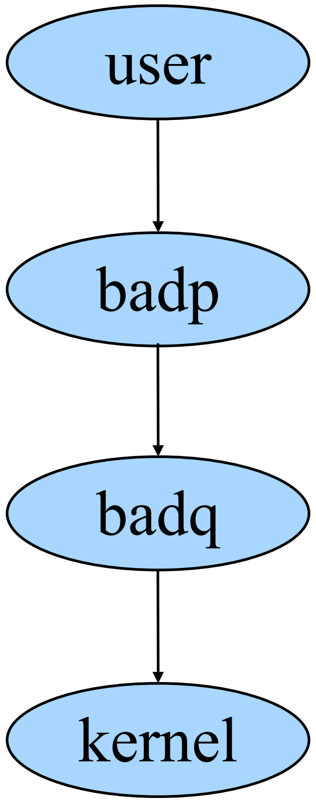
Qualifier inference example

```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```



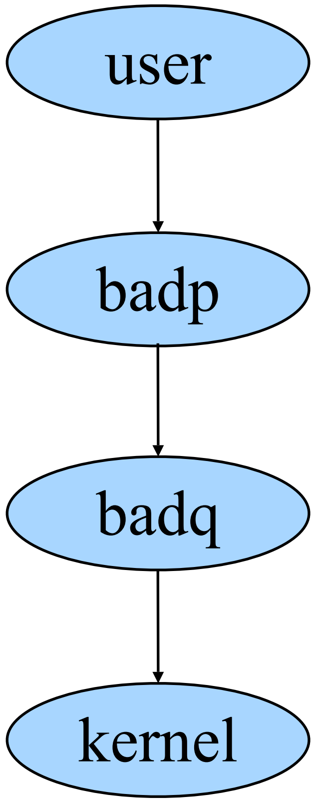
Qualifier inference example

```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```



Qualifier inference example

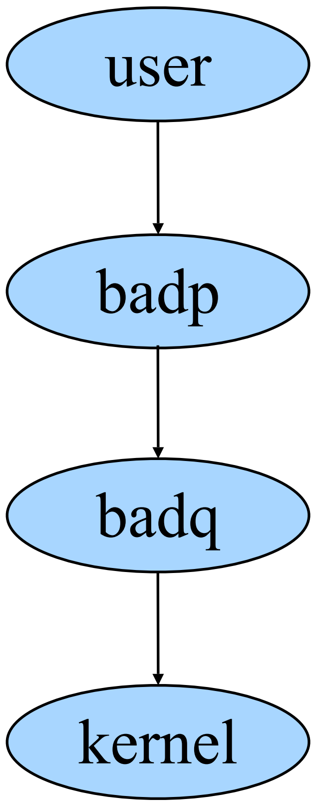
```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```



$user \leq badp \leq badq \leq kernel$

Qualifier inference example

```
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                  int len);

int bad ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}
```



$\mathit{user} \leq \mathtt{badp} \leq \mathtt{badq} \leq \mathit{kernel}$

$\mathit{user} \not\leq \mathit{kernel}$

CQUAL

- Tool for type qualifier inference and checking
- Authors extended the tool to support user and kernel qualifiers
- Ran the tool on Linux kernel source
- Limitations resulted in many false positives
- Refined tool to eliminate false positives

Context Sensitivity

```
void * helper (void *h) {
    assert h != null;
    return h;
}

int good_ioctl (void * user goodp) {
    char goodbuf[8];
    void *q = helper(goodp);
    void *b = helper(goodbuf);
    copy_from_user(b, q, 8);
}
```

- Both good and bad pointers flow through `helper()`
- `helper` should be polymorphic in qualifier:
  - `∀α void * α helper (void * α h)`
- Actual implementation involves labeling graph edges

Field Sensitivity

```
struct foo { int a; }
void sys_foo (char * user p) {
    struct foo x;
    struct foo y;
    x.a = p;
    *(y.a) = 0;
}
```

- Originally all `foo.a` were given the same qualifier
- Assigning quals to all fields takes too much memory
  - Instead do it on demand
- Unify entire structure on assignment (e.g. `x = y`)

Well-formedness Constraints

user flows down pointers

- `char *` *`user`* `a` → `char` *`user`* `*` *`user`* `a`
- *`user`* `ref(α char)` → *`user`* `ref(`*`user`* `char)`
- could also flow up pointers but not in this use case

Flowing to structure fields

- `struct foo { int a }`
- `struct foo user;` → `foo.a` **gets** *`user`*
- `struct foo * user` → `foo->a` **gets** *`user`*

Pointer/Integer Casts

```
char **p = ...;
int x = (int)p;
```

Before: α ref(α' ref(α'' char) ≤ β int
Collapses: α = α' = α'' (all ≤ β)

Treat: int **as** void *

Now: α ref(α' ref(α'' char) ≤ β ref(β' void)
Now: α ≤ β and α' = α'' = β'

Still collapses, but is more precise and (unlike before) sound.

Error clustering

Planned clustering:

- Sort errors from shortest to longest
- For each qualified variable:
  - print only one path passing through that variable

Additional clustering:

- Done manually by the line of code from which the user pointer originates

Still generates false positives

User/Kernel flag passed at runtime:

```
void tty_write (void *p, int from_user) {
    char buf[8];
    if (from_user)
        copy_from_user(buf, p, 8);
    else
        memcpy(buf, p, 8);
}
```

Still generates false positives

C type misuse:

```
void makemsg (char *buf) {
    char msg[10];
    msg[0] = READ_REGISTER;
    msg[1] = 5;
    msg[2] = buf;
    ...
```

Still generates false positives

Temporary variable reuse:

```
void good_ioctl (char * user up) {
    char buf1[10], buf2[10];
    copy_from_user(buf1, up, 10);

    up = malloc(10);
    ...
    memcpy(buf2, up, 10);
}
```

Assumptions

- Memory safe (no buffer overflows)
- Unions are used safely
- No separate compilation:
  - require whole-program-analysis for soundness
- Ignore inline assembly