

Finding User/Kernel Pointer Bugs with Type Inference

Rob Johnson
David Wagner

{rtjohnso,daw}@cs.berkeley.edu.

UC Berkeley

User/Kernel Pointer Bugs

- OS kernels cannot trust system call arguments
- Many system calls take pointers to user buffers
 - `read`
 - `write`
 - `ioctl`
 - 126 more in Linux 2.4.20
- Dereferencing unchecked user pointers is dangerous
- Attacker can:
 - Write to arbitrary kernel memory
 - Read arbitrary kernel memory
 - Cause kernel OOPS

Example: User/Kernel Pointer Bugs

```
void memset (void * buf,  
            int c, int len);
```

```
void dev_ioctl (void * p)  
{  
    memset(p, 0, 10);  
}
```

- Problem: What if `p` is `NULL`?
- In general, attacker may be able to
 - Crash kernel
 - Gain root privileges
 - Read secret data from kernel buffers

A Big Problem

- User/kernel pointer vulnerabilities have been a persistent problem in the Linux kernel
- Every kernel we checked had user/kernel pointer bugs
- Kernel developers designed easy interface for accessing user pointers
 - `copy_from_user(dest, src, len);`
 - `copy_to_user(dest, src, len);`

Our Solution: Type Qualifiers

```
void memset (void * $kernel buf,  
            int c, int len);
```

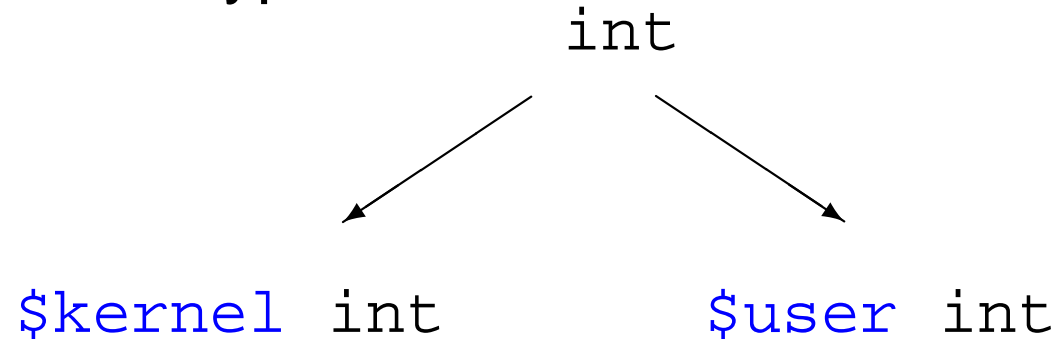
```
int dev_ioctl (void * $user p)  
{  
    memset(p, 0, 10);  
}
```

- Two types of pointers: user pointers and kernel pointers
- Refine C type system to reflect this dichotomy

Insecure programs won't typecheck

Type Qualifiers

- Qualifiers refine basic types



- “`$kernel int`” is called a *qualified type*
- Qualifiers must match in assignments:

```
$kernel int k1, k2;  
$user int u;  
k1 = k2; // OK  
k2 = u; // ERROR
```

Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

```
$kernel    int k;  
            int y;  
            int x;  
  
$user      int u;  
y = k;  
u = x;
```

Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

\$kernel	<code>int k;</code>	$Q_k = $	\$kernel
	<code>int y;</code>	$Q_y = $?
	<code>int x;</code>	$Q_x = $?
\$user	<code>int u;</code>	$Q_u = $	\$user

`y = k;`
`u = x;`

Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

\$kernel	<code>int k;</code>	$Q_k = $	\$kernel
	<code>int y;</code>	$Q_y = $?
	<code>int x;</code>	$Q_x = $?
\$user	<code>int u;</code>	$Q_u = $	\$user
<code>y = k;</code>		$Q_k = $	Q_y
<code>u = x;</code>		$Q_x = $	Q_u

Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

\$kernel	<code>int k;</code>	$Q_k =$	\$kernel
	<code>int y;</code>	$Q_y =$?
	<code>int x;</code>	$Q_x =$?
\$user	<code>int u;</code>	$Q_u =$	\$user
<code>y = k;</code>		$Q_k =$	Q_y
<code>u = x;</code>		$Q_x =$	Q_u

- Exactly one solution:

$$\mathbf{\$kernel} = Q_k = Q_y \qquad Q_x = Q_u = \mathbf{\$user}$$

Detecting Security Violations

- Programming errors yield unsolvable constraint systems

```
$kernel    int k;  
           int y;  
$user     int u;  
y = u;  
k = y;
```

Detecting Security Violations

- Programming errors yield unsolvable constraint systems

<code>\$kernel</code>	<code>int k;</code>	$Q_k = $	<code>\$kernel</code>
	<code>int y;</code>	$Q_y = $	<code>?</code>
<code>\$user</code>	<code>int u;</code>	$Q_u = $	<code>\$user</code>
<code>y = u;</code>		$Q_u = $	Q_y
<code>k = y;</code>		$Q_y = $	Q_k

$$\text{\$user} = Q_u = Q_y = Q_k = \text{\$kernel}$$

But, by definition,

$$\text{\$user} \neq \text{\$kernel}$$

NO SOLUTION

Verification vs. Bug-Finding

- Bug-finding tools
 - Few false positives
 - Miss bugs
- Verification tools
 - More false positives
 - Find all bugs
- Security requires *absence* of bugs, i.e. verification
- Theorem: Type qualifier inference is sound for memory-safe programs

Type qualifier inference gives security guarantees

Experimental Setup

- Annotated Linux kernel
 - System calls
 - User-pointer access functions
 - Common inline assembly functions
 - Dereference operator
 - ≈ 300 annotations
- Two kernel configurations
 - Full: all drivers and features enabled
 - Default: core kernel, a few common drivers
- Used CQUAL in two modes
 - Bug-finding mode: unsound, but interactive
 - Verification mode: sound, but batch processing

Experimental Results

Version	Config	Mode	Sound	Bugs	FPs
2.4.20	Full	Bug-finding	No	7	275
2.4.23	Full	Bug-finding	No	6	264
2.4.23	Default	Bug-finding	No	1	76
2.4.23	Default	Verification	Yes	4	53

- Found 17 different security vulnerabilities
- Found bugs missed by other auditing tools
- Found bugs missed by manual audits
- All but one bug confirmed exploitable
- Discovered significant “bug churn”

Steps To Verification

- Default kernel configuration
 - Fix user/kernel pointer bugs
 - Clean up kernel to avoid false positives
- Full kernel configuration
 - Fix user/kernel pointer bugs
 - Clean up kernel to avoid false positives
 - Buy RAM

Quick and Easy CQUAL

- Based on feedback from Linux Kernel Mailing List
- Ported annotations to Linux 2.6.7-rc3
- Analyzed Linux 2.6.7-rc3
- Found 7 new security vulnerabilities
- Total work: 2 days

Example: drivers/usb/w9968cf.c

```
static int
w9968cf_do_ioctl(struct w9968cf_device* cam,
                 unsigned cmd, void* arg)
{
    ...
    case VIDIOCGFBUF:
    {
        struct video_buffer* buffer =
            (struct video_buffer*) arg;

        memset( buffer, 0,
                sizeof(struct video_buffer));
    }
}
```

Example: drivers/usb/w9968cf.c

```
static int
w9968cf_do_ioctl(struct w9968cf_device* cam,
                 unsigned cmd, void* arg)
{
    ...
    case VIDIOCGFBUF:
    {
        struct video_buffer* buffer =
            (struct video_buffer*) arg;

        memset( buffer, 0,
                sizeof(struct video_buffer));
    }
}
```

Example: drivers/usb/w9968cf.c

```
static int
w9968cf_do_ioctl(struct w9968cf_device* cam,
                 unsigned cmd, void* arg)
{
    ...
    case VIDIOCGFBUF:
    {
        struct video_buffer* buffer =
            (struct video_buffer*) arg;

        memset( buffer, 0,
                sizeof(struct video_buffer));
    }
}
```

Example: drivers/usb/w9968cf.c

```
static int
w9968cf_do_ioctl(struct w9968cf_device* cam,
                 unsigned cmd, void* arg)
{
    ...
    case VIDIOCGFBUF:
    {
        struct video_buffer* buffer =
            (struct video_buffer*) arg;

        memset( buffer, 0,
                sizeof(struct video_buffer));
    }
}
```

Example: drivers/i2c/i2c-dev.c

```
int i2cdev_ioctl (unsigned int cmd,
                  unsigned long arg)
{
    struct i2c_rdwr_ioctl_data rdwr_arg;
    ...
    case I2C_RDWR:
        copy_from_user( &rdwr_arg, arg,
                       sizeof(rdwr_arg));

        ...
        copy_to_user( rdwr_arg.msgs[i].buf,
                     rdwr_pa[i].buf,
                     rdwr_pa[i].len);
```

Example: drivers/i2c/i2c-dev.c

```
int i2cdev_ioctl (unsigned int cmd,  
                 unsigned long arg)  
{  
    struct i2c_rdwr_ioctl_data rdwr_arg;  
    ...  
    case I2C_RDWR:  
        copy_from_user( &rdwr_arg, arg,  
                       sizeof(rdwr_arg));  
        ...  
        copy_to_user( rdwr_arg.msgs[i].buf,  
                    rdwr_pa[i].buf,  
                    rdwr_pa[i].len);
```

Example: drivers/i2c/i2c-dev.c

```
int i2cdev_ioctl (unsigned int cmd,
                  unsigned long arg)
{
    struct i2c_rdwr_ioctl_data rdwr_arg;
    ...
    case I2C_RDWR:
        copy_from_user( &rdwr_arg, arg,
                        sizeof(rdwr_arg));

        ...
        copy_to_user( rdwr_arg.msgs[i].buf,
                      rdwr_pa[i].buf,
                      rdwr_pa[i].len);
```

Example: drivers/i2c/i2c-dev.c

```
int i2cdev_ioctl (unsigned int cmd,
                  unsigned long  arg)
{
    struct i2c_rdwr_ioctl_data rdwr_arg;
    ...
    case I2C_RDWR:
        copy_from_user( &rdwr_arg, arg,
                        sizeof(rdwr_arg));

        ...
        copy_to_user( rdwr_arg.msgs[i].buf,
                      rdwr_pa[i].buf,
                      rdwr_pa[i].len);
```

Example: drivers/i2c/i2c-dev.c

```
$ kqual i2c-dev.i
i2c-dev.h:44 WARNING: rdwr_arg.msgs treated
                    as $user and $kernel

rdwr_arg.msgs: $kernel $user
  proto.cq:27   $user  <= *copy_from_user_arg1
i2c-dev.c:254  <= *copy_from_user_arg1
i2c-dev.c:254  <= rdwr_arg
i2c-dev.c:218  <= rdwr_arg.msgs
i2c-dev.h:44   <= &rdwr_arg.msgs->buf
i2c-dev.c:301  <= _op_deref_arg1
  proto.cq:140 <= $kernel

...
```

Related Work

- CQUAL [Foster]
- Percent-S [Shankar]
- MECA [Engler]
- **sparse** [Torvalds]
- Model checkers
 - MOPS [Chen]
 - SLAM [Ball]
 - BLAST [Henzinger]
- Lexical tools: LCLint, ITS-4, RATS, etc.

Conclusions

- Type qualifier inference
 - Finds lots of security bugs with minimal effort
 - Can scale to large programs
 - Brings us very close to program verification
- Details in paper
 - More precise type inference techniques
 - Methods for improving analysis results presentation
 - Detailed analysis of false positives
 - Verifiable programming guidelines

`http://cqual.sf.net/`

A CQUAL Tutorial

Advanced Inference Techniques

- Increased precision
 - Qualifier subtyping
 - Polymorphism
 - Field sensitivity
- Usability: Warning clustering
- Expressiveness
 - Well-formedness conditions
 - Effects

Qualifier Subtyping

- Trusted data can be used as untrusted

`$trusted` \prec `$untrusted`

- Qualifier subtyping extends to qualified type subtyping

`$trusted int` \prec `$untrusted int`

```
$trusted    int t;  
$untrusted int u;  
u = t;      // OK  
t = u;      // ERROR
```

Polymorphism

```
void * id(void *h)
{
    return h;
}
```

```
int good_ioctl(void * $user goodp)
{
    char goodbuf[8];
    void *q = id(goodp);
    void *b = id(goodbuf);

    copy_from_user(b, q, 8);
}
```

Polymorphism

```
void * id(void *h)
{
    return h;
}
```

id: void * \$user → void * \$user

```
int good_ioctl(void * $user goodp)
{
    char goodbuf[8];
    void *q = id(goodp);
    void *b = id(goodbuf);

    copy_from_user(b, q, 8);
}
```

Polymorphism

```
void * id(void *h)
{
    return h;
}
```

id: void * **\$user** → void * **\$user**

id: void * **\$kernel** → void * **\$kernel**

```
int good_ioctl(void * $user goodp)
{
    char goodbuf[8];
    void *q = id(goodp);
    void *b = id(goodbuf);

    copy_from_user(b, q, 8);
}
```

Polymorphism

```
void * id(void *h)
{
    return h;
}
```

id: void * \$user \rightarrow void * \$user

id: void * \$kernel \rightarrow void * \$kernel

id: $\forall Q$: void * Q \rightarrow void * Q

```
int good_ioctl(void * $user goodp)
{
    char goodbuf[8];
    void *q = id(goodp);
    void *b = id(goodbuf);

    copy_from_user(b, q, 8);
}
```

Field Sensitivity

```
void func(int * $kernel k, int * $user u)
{
    struct foo a, b;
    a.p = k;
    b.p = u;
}
```

- Field insensitive analysis:

$$\text{\$kernel} = Q_k = Q_p = Q_u = \text{\$user}$$

- Field sensitive analysis:

$$\text{\$kernel} = Q_k = Q_{a.p} \quad Q_{b.p} = Q_u = \text{\$user}$$

Warning Clustering

```
void func(int * $user u)
{
    int *a, *b, *c;
    a = u;
    b = a;
    c = b;
    *c = 0;
}
```

$$\text{\$user} = Q_u = Q_a = Q_b = Q_c = \text{\$kernel}$$

- Program has 4 type errors: u, a, b, c
- Cluster errors and present only one to user

Well-formedness Conditions

- Express relations between
 - Pointers and their referents
 - Structures and their fields
- Example rules
 - `$user` pointers point to `$user` data
 - `$user` structures have `$user` fields

```
void func(void * $user u)
{
    struct foo f;
    copy_from_user(&f, u, sizeof(f));
    *f.p = 0; // ERROR
}
```

Effects

- Model side-effects of functions, e.g.
 - Writes to stdout
 - Allocates memory
 - Updates a global variable
 - Takes a lock
 - Writes to certain region of memory
- Effect of a function is union of effects of all the functions it calls

Walkthrough Developing a New Analysis

Linux `__init` sections

- Linux places startup code/data in “`__init`” sections
- Kernel reclaims `__init` sections after startup
- Non-`__init` code must not use `__init` code/data
- `__init` code may safely use non-`__init` code/data
- Goal: Verify safe use of `__init` code/data

Example

```
int state __init;  
  
void reset(void) // NOT __init  
{  
    state = 0; // ERROR  
}  
  
void setup(void) __init  
{  
    reset(); // OK  
}
```

Modelling `__init` with Qualifiers

1. Create `$init` and `$noninit` qualifiers
2. Decide on any subtyping relations
3. Determine well-formedness conditions
4. Decide if effects are needed
5. Write annotations

Init subtyping

- `$noninit` code/data can be used where `$init` code/data is expected
- `$init` code/data is not always safe to use when `$noninit` code/data is expected

Init subtyping

- `$noninit` code/data can be used where `$init` code/data is expected
- `$init` code/data is not always safe to use when `$noninit` code/data is expected

`$noninit` < `$init`

Init Well-Formedness Conditions

- Structures live in the same section as their fields, i.e.
 - If a structure is `$init`, so are its fields
 - If a structure is `$noninit`, so are its fields
 - If a field is `$init`, so is the structure
 - If a field is `$noninit`, so is the structure
- Pointers may point to different sections, as long as they are used safely
 - No pointer-based well-formedness conditions

Init Effects

- Accessing the `__init` section is a side effect
- `$noninit` functions cannot have `$init` effect
- Accessing an `$init` variable gives function the `$init` effect

Init Annotations

- Only one annotation:

τ `_op_deref` (τ * Q `x`) Q `i`;

Init Example

```
struct foo F $init;
```

```
void zero(int *p) $noninit  
{  
    *p = 0;  
}
```

```
void setup(void) $init  
{  
    zero(&F.q);  
}
```