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?

\[ \text{In general, attacker may be able to} \]
\[ \text{Crash kernel} \]
\[ \text{Gain root privileges} \]
\[ \text{Read secret data from kernel buffers} \]
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

```c
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
Qualifiers refine basic types

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

- Reduce programmer work
- Derive missing annotations from context

```
$kernell
$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

```c
$kernel
    int k;
    int y;
    int x;
$kernel

$user
    int u;
$user

y = k;
u = x;
```
Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

\[
\text{\$kernel} \quad \text{int } k; \quad Q_k = \text{\$kernel} \\
\quad \text{int } y; \quad Q_y = ? \\
\quad \text{int } x; \quad Q_x = ? \\
\text{\$user} \quad \text{int } u; \quad Q_u = \text{\$user} \\
y = k; \quad Q_k = Q_y \\
u = x; \quad Q_x = Q_u
\]
Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

```c
$k$ kernel  int  $k$;  $Q_k = \$kernel$
    int  $y$;  $Q_y = \?$
    int  $x$;  $Q_x = \?$
$user$  int  $u$;  $Q_u = \$user$
$y = k$;
$u = x$;
$Q_k = Q_y$
$Q_x = Q_u$
```

- Exactly one solution:

```
$\$kernel = Q_k = Q_y     Q_x = Q_u = \$user$
```
Detecting Security Violations

Programming errors yield unsolvable constraint systems

$\text{kernel}$ int $k$;
    int $y$;
$\text{user}$ int $u$;
y = $u$;
k = $y$;
Detecting Security Violations

Programming errors yield unsolvable constraint systems

\[
\begin{align*}
& \text{$\texttt{kernel}$ int $k$;} & Q_k &= \text{$\texttt{kernel}$} \\
& \text{int $y$;} & Q_y &= \text{?} \\
& \text{$\texttt{user}$ int $u$;} & Q_u &= \text{$\texttt{user}$} \\
& y = u; & Q_u &= Q_y \\
& k = y; & Q_y &= Q_k \\
\end{align*}
\]

\[\text{$\texttt{user}$} = Q_u = Q_y = Q_k = \text{$\texttt{kernel}$}\]

But, by definition,

\[\text{$\texttt{user}$} \neq \text{$\texttt{kernel}$}\]

**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, e.g. 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 easy-to-use
  - Verification mode: sound, but less convenient
### Experimental Results

<table>
<thead>
<tr>
<th>Version</th>
<th>Config</th>
<th>Mode</th>
<th>Sound</th>
<th>Bugs</th>
<th>FPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.20</td>
<td>Full</td>
<td>Bug-finding</td>
<td>No</td>
<td>7</td>
<td>275</td>
</tr>
<tr>
<td>2.4.23</td>
<td>Full</td>
<td>Bug-finding</td>
<td>No</td>
<td>6</td>
<td>264</td>
</tr>
<tr>
<td>2.4.23</td>
<td>Default</td>
<td>Bug-finding</td>
<td>No</td>
<td>1</td>
<td>76</td>
</tr>
<tr>
<td>2.4.23</td>
<td>Default</td>
<td>Verification</td>
<td>Yes</td>
<td>4</td>
<td>53</td>
</tr>
</tbody>
</table>

- 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”
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
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));
    }
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

```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));
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);
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_argmsgs[i].buf,
                     rdwr_pa[i].buf,
                     rdwr_pa[i].len);
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);
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

...
The Extinction of User/Kernel Bugs

No more user/kernel pointer bugs — ever

**Step 1:** Eliminate current bugs and FPs in kernel

**Step 2:** Get CQUAL audits into kernel build process

Leverage kernel cleanups for *sparse*

Improve scalability of type-qualifier inference
Conclusions

- Type qualifier inference
  - Finds lots of security bugs with minimal effort
  - Can scale to large programs
  - Brings us very close to program verification

- Other applications
  - Verified election software
  - Secure sensor networks

http://cqual.sf.net/
Soundness and Security

- Asymmetry between attackers and defenders:
  - Defenders must find all bugs
  - Attackers need only one bug
Soundness Really Matters

- MECA: finds user/kernel pointer bugs, but unsound
- Have not performed direct comparison, but...
- CQUAL found 10 bugs in Linux 2.4.23
- 8 of those bugs were still in Linux 2.5.63
- MECA found 2 of them