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

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

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

Problem: What if \( p \) is \texttt{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

```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
Type Qualifiers

- Qualifiers refine basic types

```
int

$kernel int  $user int
```

- "$kernel int" is called a qualified type
- Qualifiers must match in assignments:

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

- Reduce programmer work
- Derive missing annotations from context

```c
$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

$kernell\:
\begin{array}{ll}
\text{int } & k; \\
\text{int } & y; \\
\text{int } & x;
\end{array}
\Rightarrow
\begin{array}{ll}
Q_k = & kernel \\
Q_y = & ? \\
Q_x = & ?
\end{array}

$userl:
\begin{array}{ll}
\text{int } & u;
\end{array}
\Rightarrow
\begin{array}{ll}
Q_u = & user
\end{array}

y = k;

u = x;
Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

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

$user
int u;

y = k;
u = x;

Q_k = $kernel
Q_y = ?
Q_x = ?

Q_u = $user
Q_k = Q_y
Q_x = Q_u
```
Type Qualifier Inference

- Reduce programmer work
- Derive missing annotations from context

$\text{kernel}$

```c
int k;
int y;
int x;
```

$\text{user}$

```c
int u;
```

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

Exactly one solution:

\[
\begin{align*}
$\text{kernel} &= Q_k = Q_y \\
Q_x &= Q_u = $\text{user}
\end{align*}
\]
Detecting Security Violations

Programming errors yield unsolvable constraint systems

```c
$kernel int k;
    int y;
$user int u;
y = u;
k = y;
```
Detecting Security Violations

Programming errors yield unsolvable constraint systems

```plaintext
$\text{kernal} \quad \text{int} \ k; \quad Q_k = \text{kernal}

\text{int} \ y; \quad Q_y = \text{?}

$\text{user} \quad \text{int} \ u; \quad Q_u = \text{user}

y = u; \quad Q_u = Q_y

k = y; \quad Q_y = Q_k

\text{?user} = Q_u = Q_y = Q_k = \text{kernal}
```

But, by definition,

\$\text{user} \neq \text{kernal}\$

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
  - \( \approx 300 \) 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

<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”
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
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));
    }
```
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);
Example: drivers/i2c/i2c-dev.c

```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);
```
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

...
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

\[ $\text{trusted} \prec \text{untrusted} $ \]

- Qualifier subtyping extends to qualified type subtyping

\[ $\text{trusted} \text{ int} \prec \text{untrusted} \text{ int} $ 

\begin{verbatim}
$\text{trusted}$
int $t$;

$\text{untrusted}$
int $u$;

$u = t;$  // OK

$t = u;$  // ERROR
\end{verbatim}
Polymorphism

```c
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

```c
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 \rightarrow void * $user
id:  void * $kernel \rightarrow 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

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

id: void * $user → void * $user
id: void * $kernel → void * $kernel
id: ∀Q: void * Q → void * Q

```c
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

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

- Field insensitive analysis:
  
  $$\textbf{\$kernel} = Q_k = Q_p = Q_u = \textbf{\$user}$$

- Field sensitive analysis:
  
  $$\textbf{\$kernel} = Q_k = Q_{a.p} \quad Q_{b.p} = Q_u = \textbf{\$user}$$
Warning Clustering

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

$user = Q_u = Q_a = Q_b = Q_c = $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
- $\text{user}$ pointers point to $\text{user}$ data
- $\text{user}$ structures have $\text{user}$ fields

```c
void func(void * $\text{user}$ 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

```c
int state __init;

void reset(void) // NOT __init
{
    state = 0; // ERROR
}

void setup(void) __init
{
    reset(); // OK
}
```
Modelling `__init__` with Qualifiers

1. Create $\texttt{\_init}$ and $\texttt{\_noninit}$ qualifiers
2. Decide on any subtyping relations
3. Determine well-formedness conditions
4. Decide if effects are needed
5. Write annotations
Init subtyping

- $\text{noninit}$ code/data can be used where $\text{init}$ code/data is expected

- $\text{init}$ code/data is not always safe to use when $\text{noninit}$ code/data is expected
Init subtyping

- $\text{noninit}$ code/data can be used where $\text{init}$ code/data is expected
- $\text{init}$ code/data is not always safe to use when $\text{noninit}$ code/data is expected

$\text{noninit} \prec \text{init}$
Init Well-Formedness Conditions

- Structures live in the same section as their fields, i.e.
  - If a structure is $\textit{init}$, so are its fields
  - If a structure is $\textit{noninit}$, so are its fields
  - If a field is $\textit{init}$, so is the structure
  - If a field is $\textit{noninit}$, 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:

\[ \tau \ _{\text{op\_deref}} (\tau \ * \ Q \ x) \ Q ; \]
struct foo F $init;

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

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