Security types to the rescue

David Wagner and Rob Johnson
{daw, rtjohnson}@cs.berkeley.edu

University of California, Berkeley
Problem statement

- Attackers may feed us untrustworthy inputs
- We must be careful to never trust those inputs, lest terrible things happen

How do we ensure proper **input validation**?
Input validation is tricky

- All untrusted data must get validated somewhere on the path from source to sink
- Sources of untrusted data can be far from the sinks
Strategy #1: Validate at the sources

A good choice when: proper validation depends on where data came from, rather than how it will be used.
Strategy #2: Validate at the sinks

A good choice when:
proper validation depends on how data will be used,
rather than where it came from
A hybrid strategy

This is what results when:

- Developer Dave validates at the sources
- Coder Courtney validates at the sinks
- Dave & Courtney work together
Challenges in input validation

There are many data flow paths through the program that require checks.

*How can we be sure we got them all?*
In this talk:

Q: How can we verify complete input validation?
Overview

In this talk:

Q: How can we verify complete input validation?

A: Type systems
Overview

In this talk:

Q: How can we verify complete input validation?

A: Type systems

Defn: Security types are types that record security claims.
Static taint analysis

The idea:
- Extend the C type system with type qualifiers
- Qualified types express security annotations; e.g., $tainted char * is an untrusted string
- Type checking enforces safe usage
- Type inference reduces the annotation burden

Benefits:
- verify complete input validation
- eliminate error-prone mental “bookkeeping”
Spot the bug

```c
char *s;
s = read_from_network();
syslog(s);
```
Spot the bug

char *s;
s = read_from_network();
syslog(s);
Spot the bug

```c
char *s;
s = read_from_network();  // untrusted source
syslog(s);                // trusting sink
```
Spot the bug

```c
char *s;
s = read_from_network();    // untrusted source
syslog(s);                  // trusting sink
```

**Format string bug!**
`syslog()` trusts its first argument.
Types, and the example

```c
void printf($untainted char *, ...);
$tainted char * read_from_network(void);

char *s;
s = read_from_network();
syslog(s);
```

where $\text{untainted } T \leq \text{tainted } T$
Types, and the example

void printf($untainted char *, ...);
$tainted char * read_from_network(void);

\[ \rightarrow \text{a trust annotation} \]

char *s;
$s = \text{read\_from\_network}();$
syslog(s);

where $\text{$untainted } T \leq \text{$tainted } T$
Types, and the example

```c
void printf($untainted char *, ...);
$tainted char * read_from_network(void);

char *s;
s = read_from_network();
syslog(s);
```

where $\text{untainted } T \leq \text{tainted } T$
Types, and the example

```c
void printf($untainted char *, ...);
$tainted char * read_from_network(void);

char /* $tainted */ *s;
s = read_from_network();
syslog(s);

where $untainted T \leq $tainted T```

Security types to the rescue – p. 11
Types, and the example

```c
void printf($untainted char *, ...);
$tainted char * read_from_network(void);

char /* $tainted */ *s;    // an inferred type
s = read_from_network();
syslog(s);

where $untainted T \leq $tainted T
```
Types, and the example

```c
void printf($untainted char *, ...);
$tainted char * read_from_network(void);

char /* $tainted */ *s;
s = read_from_network();
syslog(s);

→ Doesn’t type-check! A security hole.

where $untainted T \leq $tainted T
```
Case study #1: Format string bugs

- Successful at finding bugs in real programs
- Cost: 10–15 minutes per application
Case study #2: the Linux kernel

Risk: TOCTTOU bugs
- Some syscalls pass values by reference, rather than by value

Current approach: manual emulation of pass-by-copy
- Problematic; it’s easy to forget to make a copy

Types can help...
int main ()
{
    struct foo *p;
    ...
    ioctl (fd, SIOCGFOO, p);
    ...
}

int dev_ioctl (int cmd, long arg)
{
    struct foo *q = mkfoo();
    ...
    copy_to_user (arg, q, n);
    return 0;
}
int main ()
{
    struct foo *p;
    ...
    ioctl (fd, SIOCGFOO, p);
    ...
}

int dev_ioctl (int cmd, long arg)
{
    struct foo *q = mkfoo();
    ...
    memcpy (arg, q, n);
    return 0;
}
Type qualifiers to the rescue

The idea:
- Annotate all pointers from user-space as $user
- Annotate all kernel pointers as $kernel
- Only allow dereferencing of $kernel pointers
- Use type qualifier inference (CQUAL)
Example: Type checking

struct foo * $kernel mkfoo(void);
int memcpy(void * $kernel dst,
    void * $kernel src, size_t n);

int dev_ioctl (int cmd, long $user arg)
{
    struct foo * $kernel q = mkfoo();
    ...
    memcpy (arg, q, n);  \[Doesn't type-check!\]
    return 0;
}

Security types to the rescue – p. 17
Example: Type inference

```c
struct foo * $kernel mkfoo(void);
int memcpy(void * $kernel dst,
            void * $kernel src, size_t n);

int dev_ioctl (int cmd, long $user arg)
{
    struct foo * /* $kernel */ q = mkfoo();
    ...
    memcpy (arg, q, n);
    return 0;
}
```

$q$’s type is inferred automatically

→ Doesn’t type-check!
Example: Type inference

struct foo * $kernel mkfoo(void);
int memcpy(void * dst, void * src, size_t n)
{ while (n-- > 0) *dst++ = *src++; }

memcpy()’s signature is inferred automatically

int dev_ioctl (int cmd, long $user arg)
{
    struct foo * /* $kernel */ q = mkfoo();
    ...
    memcpy (arg, q, n);
    return 0;
}

q’s type is inferred automatically

Doesn’t type-check!
Experience with user-kernel bugs

- Analyzed 1259 files in Linux 2.4.18
- CQUAL reported errors in 23 files
- One error was genuine, exploitable bug
- Fixed in Linux 2.4.19

Recent changes to CQUAL
- Improve bug-finding power
  (pointeral and structureal constraints)
- Reduce false positive rate
  (polymorphic type inference, separating structure instances)

Results forthcoming
Spot the bug

static int joydev_ioctl(struct inode *inode, struct file *file,
    unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
    case JSIOCSAXMAP:
        if (copy_from_user((__u8 *) arg, joydev->abspam,
            sizeof(__u8) * ABS_MAX))
            return -EFAULT;
        for (i = 0; i < ABS_MAX; i++) {
            if (joydev->abspam[i] > ABS_MAX) return -EINVAL;
            joydev->absmap[joydev->abspam[i]] = i;
        }
    }
}
The general picture

**Principle:** Objects with different security properties should receive different types.

Values under attacker control receive a $tainted$ type:
- Untrusted sources are marked $tainted$
- Anything that depends on a $tainted$ value is also marked $tainted$
- Input validation breaks the propagation of $tainted$
- A $tainted$ value at a trusting sink is an error
Concluding thoughts

- **Security types**: a disciplined style of programming
  - An *enforceable* discipline
  - Detects errors of omission, but not of commission; type system won’t let you forget to validate inputs (but did you do the *right* validation checks?)
- Type systems are useful for security engineering
- You can try this at home