# Security via Type Qualifiers

(based on J. Foster's lecture
at 2004 summer school on software security)

**2004 SIGPL 여름학교**
**2004. 08. 12**

조 장 우
부산외대 컴퓨터공학부

## Introduction

- Ensuring that software is secure is hard
- Standard practice for software quality:
  - Testing
    - Make sure program runs correctly on set of inputs
  - Code auditing
    - Convince yourself and others that your code is correct

## Drawbacks to Standard Approaches

- Difficult
- Expensive
- Incomplete

- A malicious adversary is trying to exploit anything you miss!



## Tools for Security

- What more can we do?
  - Build tools that analyze source code
    - Reason about all possible runs of the program
  - Check limited but very useful properties
    - Eliminate categories of errors
  - Develop programming models
    - Avoid mistakes in the first place
    - Encourage programmers to think about security

## Tools Need Specifications

- Goal: Add specifications to programs
  In a way that…
  - Programmers will accept
    - Lightweight
  - Scales to large programs
  - Solves many different problems



## Type Qualifiers

- Extend standard type systems (C, Java, ML)
  - Programmers already use types
  - Programmers understand types
  - Get programmers to write down a little more…

| | |
|---|---|
| const int | ANSI C |
| ptr(tainted char) | Format-string vulnerabilities |
| kernel ptr(char) → char | User/kernel vulnerabilities |

## Application: Format String Vulnerabilities

- I/O functions in C use format strings

```
printf("Hello!");                    Hello!
printf("Hello, %s!", name);          Hello, name!
```

- Instead of

```
printf("%s", name);
```

Why not

```
printf(name);                ?
```



## Format String Attacks

- Adversary-controlled format specifier

```
name := <data-from-network>
printf(name);    /* Oops */
```

  - Attacker sets name = "%s%s%s" to crash program
  - Attacker sets name = "...%n..." to write to memory
    - Yields (often remote root) exploits

- Lots of these bugs
  - New ones weekly on bugtraq mailing list
  - Too restrictive to forbid variable format strings

## Basic Idea to check Format String Vulnerabilities

- Treat all program inputs that could be controlled by attacker as tainted.
- Track the propagation of tainted data through the program operations
- Mark any variable that is assigned a value from tainted data as tainted
- If tainted data is used as a format string on some execution path, we detect an Format String Vulnerabilities



## Using Tainted and Untainted

- Add qualifier annotations

```
int printf(untainted char *fmt, ...)
tainted char *getenv(const char *)
```

tainted = may be controlled by adversary

untainted = must not be controlled by adversary

## Subtyping

```
void f(tainted int);
untainted int a;
f(a);
```

OK

```
void g(untainted int);
tainted int b;
g(b);
```

Error

f accepts tainted or untainted data

g accepts only untainted data

$\text{untainted} \leq \text{tainted}$

$\text{tainted} \not\leq \text{untainted}$

$\text{untainted} < \text{tainted}$



## Extending the Qualifier Order to Types

$$\frac{Q \leq Q'}{\text{bool}^{Q} \leq \text{bool}^{Q'}} \qquad \frac{Q \leq Q'}{\text{int}^{Q} \leq \text{int}^{Q'}}$$

## Subtyping on Function Types

- What about function types?

$$\frac{?}{qt1' \rightarrow^{Q} qt2' \leq qt1 \rightarrow^{Q'} qt2}$$

- Recall: S is a subtype of T if an S can be used anywhere a T is expected
  - When can we replace a call "f x" with a call "g x"?



## Replacing "f x" by "g x"

- When is $qt1' \rightarrow^{Q'} qt2' \leq qt1 \rightarrow^{Q} qt2$ ?
- Return type:
  - We are expecting qt2 (f's return type)
  - So we can only return *at most* qt2
  - $qt2' \leq qt2$
- Example: A function that returns tainted can be replaced with one that returns untainted

## Replacing "f x" by "g x" (cont'd)

- When is $qt1' \rightarrow^{Q'} qt2' \leq qt1 \rightarrow^{Q} qt2$ ?
- Argument type:
  - We are supposed to accept $qt1$ (f's argument type)
  - So we must accept *at least* $qt1$
  - $qt1 \leq qt1'$
- Example: A function that accepts untainted can be replaced with one that accepts tainted



## Subtyping on Function Types

$$\frac{qt1' \leq qt1 \quad qt2 \leq qt2' \quad Q \leq Q'}{qt1 \rightarrow^{Q} qt2 \leq qt1' \rightarrow^{Q'} qt2'}$$

- We say that $\rightarrow$ is
  - *Covariant* in the range (subtyping dir the same)
  - *Contravariant* in the domain (subtyping dir flips)

## Subtyping References

- The *wrong* rule for subtyping references is

$$\frac{Q \le Q' \quad qt \le qt'}{ref^{Q}\ qt \le ref^{Q'}\ qt'}$$

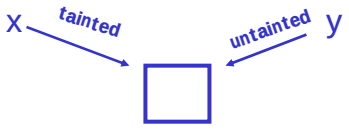
- Counterexample

```
let x = ref 0^untainted in
  let y = x in
    y := 3^tainted;
    check(untainted, !x)        oops!
```



## You've Got Aliasing!

- We have multiple names for the same memory location
  - But they have different types
  - *And* we can **write** into memory at different types



## Solution #1: Java's Approach

- Java uses this subtyping rule
  - If S is a subclass of T, then S[] is a subclass of T[]
- Counterexample:
  - Foo[] a = new Foo[5];
  - Object[] b = a;
  - b[0] = new Object(); // forbidden at runtime
  - a[0].foo(); // …so this can't happen



## Solution #2: Purely Static Approach

- Reason from rules for functions
  - A reference is like an object with two methods:
    - get : unit → qt
    - set : qt → unit
  - Notice that qt occurs both co- and contravariantly
- The right rule:

$$\frac{Q \le Q' \quad qt \le qt' \quad qt' \le qt}{ref^{Q}\ qt \le ref^{Q'}\ qt'} \quad \text{or} \quad \frac{Q \le Q' \quad qt = qt'}{ref^{Q}\ qt \le ref^{Q'}\ qt'}$$

## Demo of cqual

http://www.cs.berkeley.edu/~jfoster

## Framework

- Pick some qualifiers
  - and relation (partial order) among qualifiers

untainted int < tainted int

kernel ptr < user ptr

- Add a few explicit qualifiers to program
- Infer remaining qualifiers
  - and check consistency

## Type Qualifier Inference

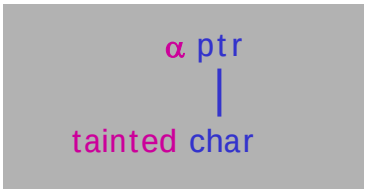
- Two kinds of qualifiers
  - Explicit qualifiers: tainted, untainted, ...
  - Unknown qualifiers: $\alpha_0$, $\alpha_1$, ...
- Program yields constraints on qualifiers

  tainted $\leq \alpha_0$  $\alpha_0 \leq$ untainted
- Solve constraints for unknown qualifiers
  - Error if no solution

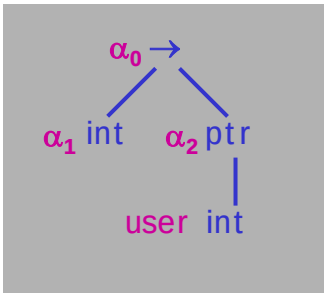


## Adding Qualifiers to Types

ptr(tainted char)

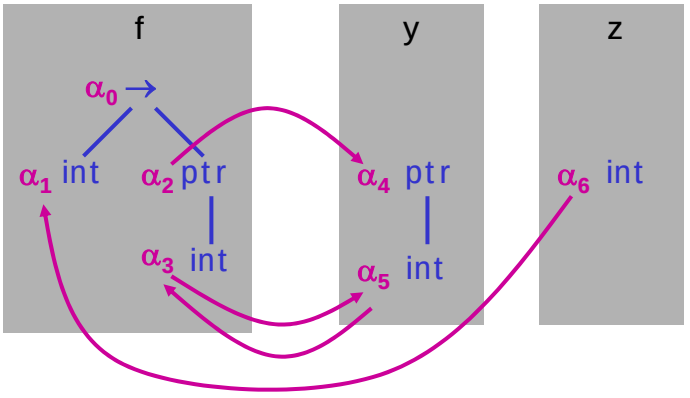



int → user ptr(int)



## Constraint Generation

ptr(int) f(x : int) = { … }  y := f(z)




$\alpha_6 \leq \alpha_1$

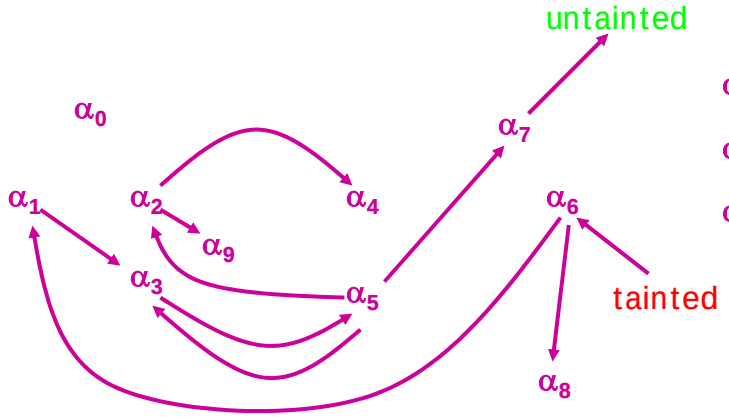
$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$



## Constraints as Graphs

Key idea: programs → constraints → graphs



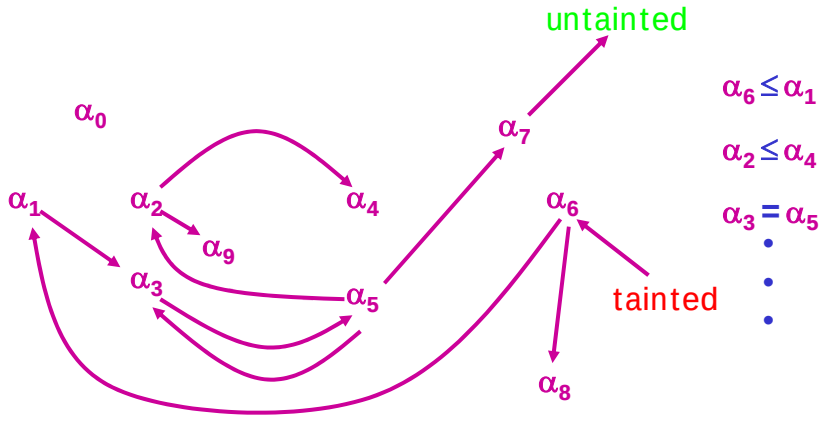

$\alpha_6 \leq \alpha_1$

$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

## Satisfiability via Graph Reachability

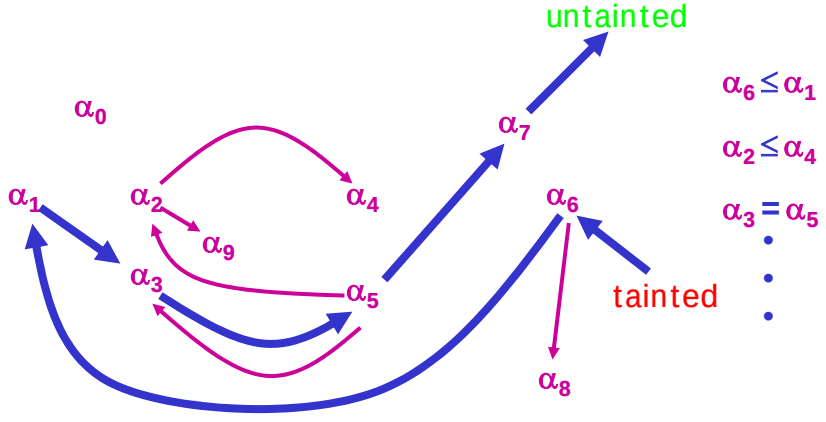
Is there an inconsistent path through the graph?



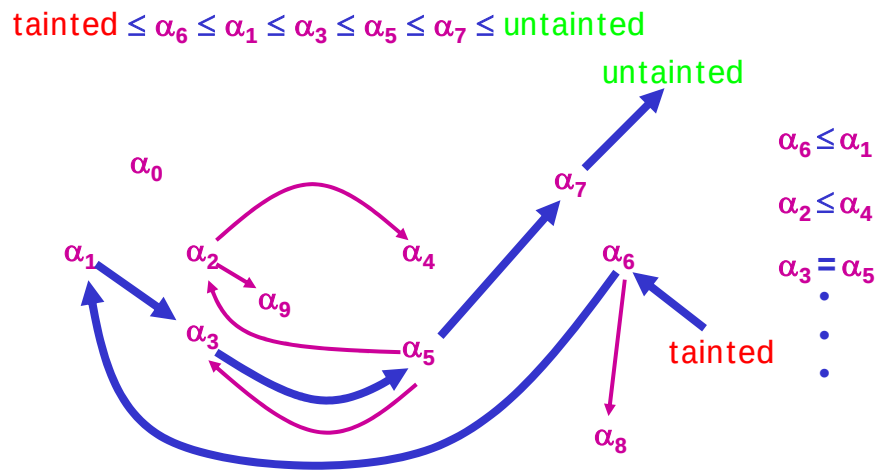



## Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



## Satisfiability via Graph Reachability






## Satisfiability in Linear Time

- Initial program of size n
  - Fixed set of qualifiers tainted, untainted, ...
- Constraint generation yields O(n) constraints
  - Recursive abstract syntax tree walk
- Graph reachability takes O(n) time
  - Works for semi-lattices, discrete p.o., products

## The Story So Far...

- Type qualifiers as subtyping system
  - Qualifiers live on the standard types
  - Programs → constraints → graphs
- Useful for a number of real-world problems
- Followed by: Experiments



## Experiment: Format String Vulnerabilities

- Analyzed 10 popular unix daemon programs
  - Annotations shared across applications
    - One annotated header file for standard libraries
- Found several known vulnerabilities
  - Including ones we didn't know about
- User interface critical

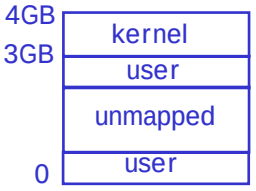
## Results: Format String Vulnerabilities

| Name | Warn | Bugs |
|---|---|---|
| identd-1.0.0 | 0 | 0 |
| mingetty-0.9.4 | 0 | 0 |
| bftpd-1.0.11 | 1 | 1 |
| muh | 12 | 1 |
| cfengine-1.5.4 | 5 | 3 |
| imapd-4.7c | 0 | 0 |
| ipopd-4.7c | 0 | 0 |
| mars_nwe-0.99 | 0 | 0 |
| apache-1.3.12 | 0 | 0 |
| openssh-2.3.0p1 | 0 | 0 |



## Experiment: User/kernel Vulnerabilities (Johnson + Wagner 04)

- In the Linux kernel, the kernel and user/mode programs share address space




  - The top 1GB is reserved for the kernel
  - When the kernel runs, it doesn't need to change VM mappings
    - Just enable access to top 1GB
    - When kernel returns, prevent access to top 1GB

## An Attack

- Suppose we add two new system calls

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

- Suppose a user calls getint(buf)
  - Well-behaved program: buf points to user space
  - Malicious program: buf points to unmapped memory
  - Malicious program: buf points to kernel memory
    - We've just written to kernel space! Oops!



## Another Attack

- Can we compromise security with setint(buf)?
  - What if buf points to private kernel data?
    - E.g., file buffers
  - Result can be read with getint

## The Solution: copy_from_user, copy_to_user

- Our example should be written

```
int x;
void sys_setint(int *p) { copy_from_user(&x, p, sizeof(x)); }
void sys_getint(int *p) { copy_to_user(p, &x, sizeof(x)); }
```

- These perform the required safety checks
  - On their user pointer arguments



## It's Easy to Forget These

- Pointers to kernel and user space look the same
  - That's part of the point of the design
- Linux 2.4.20 has 129 syscalls with pointers to user space
  - All 129 of those need to use copy_from/to
  - The ioctl implementation passes user pointers to device drivers (without sanitizing them first)
- The result: Hundreds of copy_from/_to
  - One (small) kernel version: 389 from, 428 to
  - And there's no checking

## User/Kernel Type Qualifiers

- We can use type qualifiers to distinguish the two kinds of pointers
  - kernel -- This pointer is under kernel control
  - user -- This pointer is under user control
- Subtyping kernel < user
  - It turns out copy_from/copy_to can accept pointers to kernel space where they expect pointers to user space



## Type Signatures

- We add signatures for the appropriate fns:

```
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)); }
```

Lives in kernel

OK

OK

Error

## Qualifiers and Type Structure

- Consider the following example:

```
void ioctl(void *user arg) {
  struct cmd { char *datap; } c;
  copy_from_user(&c, arg, sizeof©);
  c.datap[0] = 0;    // not a good idea
}
```

- The pointer arg comes from the user
  - So datap in c also comes from the user
  - We shouldn't deference it without a check



## Well-Formedness Constraints

- Simpler example

  char **user p;
  - Pointer p is under user control
  - Therefore so is *p
- We want a rule like:
  - In type $\text{ref}^{\text{user}}$ (Q s), it must be that $Q \leq \text{user}$
  - This is a *well-formedness* condition on types

## Well-Formedness Constraints

- As a type rule

$$\frac{\text{|--wf (Q' s)} \quad Q' \leq Q}{\text{|--wf ref}^{Q}\text{ (Q' s)}}$$

  - We implicitly require all types to be well-formed

- But what about other qualifiers?
  - Not all qualifiers have these structural constraints
  - Or maybe other quals want $Q \leq Q'$



## Well-Formedness Constraints

- Similar constraints for struct types

$$\frac{\text{For all i,} \quad \text{|--wf (Qi si)} \quad Q \leq Qi}{\text{|--wf struct}^{Q}\text{ (Q1 s1, ..., Qn sn)}}$$

  - Again, can specify this per-qualifier

## A Tricky Example

*int copy_from_user(<kernel>, <user>, <size>);*

```
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
                 unsigned long arg) {
  …case I2C_RDWR:
      if (copy_from_user(&rdwr_arg,
                         (struct i2c_rdwr_iotcl_data *) arg,
                         sizeof(rdwr_arg)))
        return -EFAULT;
      for (i = 0; i < rdwr_arg.nmsgs; i++) {
        if (copy_from_user(rdwr_pa[i].buf,
                           rdwr_arg.msgs[i].buf,
                           rdwr_pa[i].len)) {
              res = -EFAULT; break;
    } }
```



## A Tricky Example

*int copy_from_user(<kernel>, <user>, <size>);*

```
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
                 unsigned long arg) {
  …case I2C_RDWR:
      if (copy_from_user(&rdwr_arg,
                         (struct i2c_rdwr_iotcl_data *) arg,
                         sizeof(rdwr_arg)))
        return -EFAULT;
      for (i = 0; i < rdwr_arg.nmsgs; i++) {
        if (copy_from_user(rdwr_pa[i].buf,
                           rdwr_arg.msgs[i].buf,
                           rdwr_pa[i].len)) {
              res = -EFAULT; break;
    } }
```

## A Tricky Example

*int copy_from_user(<kernel>, <user>, <size>);*

```
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
                 unsigned long arg) {
  …case I2C_RDWR:
      if (copy_from_user(&rdwr_arg,
                         (struct i2c_rdwr_iotcl_data *) arg,
                         sizeof(rdwr_arg)))
        return -EFAULT;
      for (i = 0; i < rdwr_arg.nmsgs; i++) {
        if (copy_from_user(rdwr_pa[i].buf,
                           rdwr_arg.msgs[i].buf,
                           rdwr_pa[i].len)) {
            res = -EFAULT; break;
    } }
```

user

OK



## A Tricky Example

*int copy_from_user(<kernel>, <user>, <size>);*

```
int i2cdev_ioctl(struct inode *inode, struct file *file, unsigned cmd,
                 unsigned long arg) {
  …case I2C_RDWR:
      if (copy_from_user(&rdwr_arg,
                         (struct i2c_rdwr_iotcl_data *) arg,
                         sizeof(rdwr_arg)))
        return -EFAULT;
      for (i = 0; i < rdwr_arg.nmsgs; i++) {
        if (copy_from_user(rdwr_pa[i].buf,
                           rdwr_arg.msgs[i].buf,
                           rdwr_pa[i].len)) {
            res = -EFAULT; break;
    } }
```

user

OK

Bad

## Experimental Results

- Ran on two Linux kernels
  - 2.4.20 -- 11 bugs found
  - 2.4.23 -- 10 bugs found
- Needed to add 245 annotations
  - Copy_from/to, kmalloc, kfree, …
  - All Linux syscalls take user args (221 calls)
    - Could have be done automagically (All begin with sys_)



## Observations

- Several bugs persisted through a few kernels
  - 8 bugs found in 2.4.23 that persisted to 2.5.63
  - An unsound tool, MECA, found 2 of 8 bugs
  - ==> Soundness matters!
- Of 11 bugs in 2.4.23…
  - 9 are in device drivers
  - Good place to look for bugs!
  - Note: errors found in “core” device drivers
    - (4 bugs in PCMCIA subsystem)

## Observations

- Lots of churn between kernel versions
  - Between 2.4.20 and 2.4.23
    - 7 bugs fixed
    - 5 more introduced



## Conclusion

- Type qualifiers are specifications that…
  - Programmers will accept
    - Lightweight
  - Scale to large programs
  - Solve many different problems
- In the works: ccqual, jqual, Eclipse interface