CS 5430

Information-Flow Control

Prof. Clarkson
Spring 2016

Review: Information flow

- Secure information flow: no unauthorized flow of information is possible
  - Function + that **combines** security labels: $\ell_1 + \ell_2$ is label of information derived from ℓ_1 and ℓ_2
  - Relation $\rightarrow$ that **specifies what flows are allowed**: if $\ell_1 \rightarrow \ell_2$ then information from label ℓ_1 may flow to ℓ_2
- A system has **secure information flow** iff its execution never causes an information flow that violates $\rightarrow$
  - Suppose $f(a_1, ..., a_n)$ flows to b...
  - If b's label is **static**, then $L(a_1)+...+L(a_n) \rightarrow L(b)$ must hold
  - If b's label is **dynamic**, then L(b) must be updated such that $L(a_1)+...+L(a_n) \rightarrow L(b)$ holds

Review: Security conditions

- **Noninterference:** Commands of high security users have no effect on observations of low security users
  - That's Goguen & Meseguer's original definition
  - Many other conditions go by the same name
- **Noninference:** Anything that could happen in the presence of high events could also happen without them, so nothing can be inferred about their occurrence
- **Separability:** System behaves as though low and high parts are physically separated into two pieces (a simulated *airgap*)

Information-flow control

Today: enforcement mechanisms for secure information flow

- Dynamic (run-time): taint tracking
- Static (compile-time): type system

DYNAMIC INFORMATION-FLOW CONTROL (DIFC)

Information flow in web systems

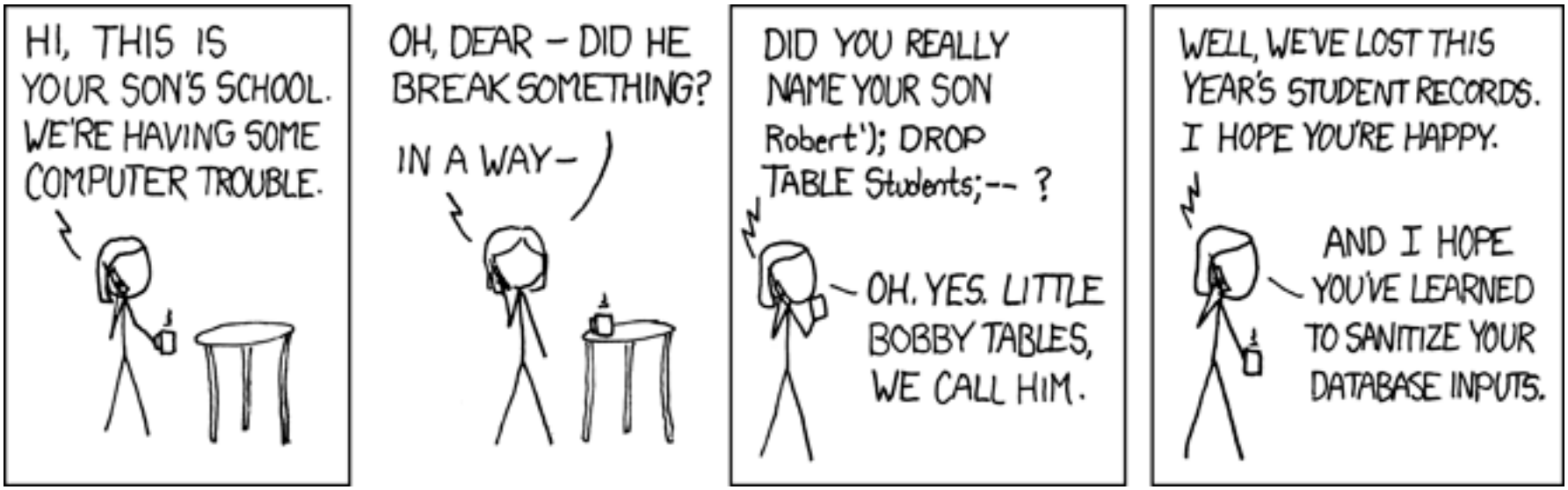
- User doesn't have account on OS
- Script has greater privileges on OS than user
  - write to disk
  - start new processes
  - etc.

Information flow in web systems

Injection attacks: exploit script's privileges to run code by providing unusual inputs

- Script injection:
  - script calls **`system("ls " + request);`**
  - malicious user request is **`"; rm -rf *"`**
- SQL injection:
  - script calls **`sql_query("select ... where name = " + request)`**
  - malicious user request is **`"...; drop table"`**

Information flow in web systems



Information flow in web systems

Defense: input validation or sanitization

- Validation: check whether input is well-formed
- Sanitization: transform input to guarantee well-formedness
- A perfect defense would require characterizing benign vs. malicious inputs (HARD)
- Less perfect: at least ensure that program always checks input, even if the check/transformation is imperfect
  - Programming language can help!
  - **Perl,** Ruby, PHP, Python, Java extension, …

Perl

[`perldoc perlsec`] (Perl 5 ca. 2011):

"You may not use data derived from outside your program to affect something else outside your program—at least, not by accident."

- information-flow policy
- integrity policy

Perl taint tracking

Data are either...

- tainted:
  - derived from outside program
  - e.g.,
    - command line arguments (`$ARGV[i]`)
    - environment variables (hence CGI script)
    - file input (hence sockets)
- untainted:
  - derived only from inside, or
  - **validated**

Perl's taint policy

- Tainted data may not be used directly or indirectly in any command that
  - invokes a subshell (i.e., gets system access), or
  - modifies a file or process
- So "tainted → outside" is a prohibited flow
- e.g., **`system ( ... $ARGV[1] ...)`** is not permitted
  - if encountered in taint mode (**`perl -T`**), halts with error **`"Insecure dependency in system"`**
  - dynamic (run-time) checking
- Helps defend against injection attacks: if programmer forgets to validate, script halts

Perl's validation mechanism

- Validation: match against a regular expression
  - Pattern match: **`$x =~ /R/`** matches value of variable $x against regular expression R
    - R may contain parenthesized expressions
    - if match succeeds, each such expression bound to special variable $1, $2, ...
  - e.g., $ARGV[2] =~ /$([^;]*)/
    - matches command-line argument
    - against regular expression that means "everything up to the first semi-colon"
    - and binds all of that to $1
- Special variables are always untainted: a form of declassification

Perl's validation mechanism

Q: Does validation by pattern matching guarantee benign values?

A: No.

- Have to get the pattern matching right
- Maybe not even possible to get it right!

Implementation of taint tracking

- Keep taint bit associated with each variable
- **Assignment statement** propagates taint, e.g.,
  - suppose statement is `$x = $y + $z;`
  - if either `$y` or `$z` is tainted, then `$x` becomes tainted too
- **Function call** checks or causes taint
  - suppose call is `f(e)`
  - if `f` is a function that affects the outside world, then `e` must be untainted
    - e.g., `system` or `write`, but (for sake of convenience?) not `print`
    - if `e` is tainted, then abort
  - if `f` is a function that is affected by the outside world, then return value is tainted
    - e.g., `read`

Implementation of taint tracking

- A curiosity: **if statements**
- Implementation doesn't keep track of whether guard is tainted
- Legal, despite policy of no "indirect" aka "implicit" flow:

```
if (read(f1) == "1")
   write(f2, "1");
else
   write(f2, "0");
```

- In fact, all purely dynamic enforcement of information flow suffers from this defect
  - Combined with some *static analysis* and *rewriting* it's possible to detect *implicit flow*
  - Advantage of dynamic enforcement: programmers write code in standard languages

Other DIFC mechanisms

- RIFLE (ISA) [Vachharajani et al. 2004]
- HiStar (OS) [Zeldovich et al. 2006]
- Trishul (JVM) [Nair et al. 2008]
- TaintDroid (Android) [Enck et al. 2010]
- LIO (Haskell) [Stefan et al. 2011]
- …

STATIC INFORMATION-FLOW CONTROL

Program certification

- Does program satisfy information-flow policy?
  - [Denning and Denning 1977]
  - Programmer provides annotations in source code
  - Compiler analyzes code, rejects program if policy could be violated
  - Helps programmers and security analysts review for security
  - In principle, end users could compile source code?
- Research languages that use this idea:
  - Jif [Myers 1999] Java + Information Flow (originally JFlow)
  - FlowCaml [Simonet 2003] OCaml + Information Flow
  - Aura, PCML5, Fine, ...

Program certification

- Does program satisfy information-flow policy?
  - [Denning and Denning 1977]
  - Programmer provides annotations in source code
  - Compiler analyzes code, rejects program if policy could be violated
  - Helps programmers and security analysts review for security
  - In principle, end users could compile source code?
- Research languages that use this idea:
  - Jif [Myers 1999] Java + Information Flow (…ow)
  - FlowCaml [Simonet 2003] OCaml + Infor…
  - Aura, PCML5, Fine, …

Jif

```
class passwordFile authority(root) {
  public boolean
    check (String user, String password)
    where authority(root) {
      // Return whether password is correct
      boolean match = false;
      try {
        for (int i = 0; i < names.length; i++) {
          if (names[i] == user &&
          passwords[i] == password) {
            match = true;
            break;
          }
        }
      }
        catch (NullPointerException e) {}
        catch (IndexOutOfBoundsException e) {}
      return declassify(match, {user; password});
  }
  private String [ ] names;
  private String { root: } [ ] passwords;
}
```

Jif

```
class passwordFile authority(root) {
    public boolean
      check (String user, String password)
      where authority(root) {
        // Return whether password is correct
        boolean match = false;
        try {
            for (int i = 0; i < names.length; i++) {
                if (names[i] == user &&
                passwords[i] == password) {
                    match = true;
                    break;
                }
            }
        }
            catch (NullPointerException e) {}
            catch (IndexOutOfBoundsException e) {}
        return declassify(match, {user; password});
    }
    private String [ ] names;
    private String { root: } [ ] passwords;
}
```

Security type: only `root` may learn information in this field

Jif

```
class passwordFile authority(root) {
  public boolean
    check (String user, String password)
    where authority(root) {
      // Return whether password is correct
      boolean match = false;
      try {
        for (int i = 0; i < names.length; i++) {
          if (names[i] == user &&
          passwords[i] == password) {
            match = true;
            break;
          }
        }
      }
      catch (NullPointerException e) {}
      catch (IndexOutOfBoundsException e) {}
    return declassify(match, {user; password});
  }
  private String [ ] names;
  private String { root: } [ ] passwords;
}
```

Declassification: okay to leak whether password matches

Jif

```
class passwordFile authority(root) {
  public boolean
    check (String user, String password)
    where authority(root) {
      // Return whether password is correct
      boolean match = false;
      try {
        for (int i = 0; i < names.length; i++) {
          if (names[i] == user &&
          passwords[i] == password) {
            match = true;
            break;
          }
        }
      }
        catch (NullPointerException e) {}
        catch (IndexOutOfBoundsException e) {}
      return declassify(match, {user; password});
  }
  private String [ ] names;
  private String { root: } [ ] passwords;
}
```

Authority:
this class is
trusted by `root`

Jif type checking

- Variables (fields, methods, etc.) may have additional label as part of their type, e.g., `int {lbl} x;`
- Label constrains information flow to and from variable
  - **reader label**: `alice -> bob, charlie`
    - Alice owns this constraint; her permission required to violate it
    - Alice permits the information to flow **to** Bob and Charlie
    - On previous slide: `root:` is short for `root -> root`
  - **writer label:** `alice <- bob, charlie`
    - Alice owns this constraint; her permission required to violate it
    - Alice permits the information to flow **from** Bob and Charlie
  - can have multiple such constraints as part of label
  - can read these arrows as the may flow relation →
  - **Decentralized label model (DLM)** [Myers and Liskov 1997]

Jif type checking

Jif type checking based on VSI type system
[Volpano, Smith, and Irvine 1996]

Geoffrey Smith (Cornell PhD 1991)

Security types

Secret variables vs. public variables

- i.e., high vs. low security
- can combine with usual types (int, bool, etc.)
- can combine with integrity, but just confidentiality for today

Leakage

- Suppose s is a secret variable and p is a public variable
- Subjects cleared at a level may observe values of variables

Do the following programs leak information?

```
1. p := p + s
2. s := p
3. p := s; p := 1
4. if (s mod 2) = 0
     then p := 0 else p := 1
5. while s != 0 do { //nothing }
```

Leakage

- Suppose s is a secret variable and p is a public variable
- Subjects cleared at a level may observe values of variables

Do the following programs leak information?

```
1. p := p + s
2. s := p
3. p := s; p := 1
4. if (s mod 2) = 0
     then p := 0 else p := 1
5. while s != 0 do { //nothing }
```

Leakage

- Suppose s is a secret variable and p is a public variable
- Subjects cleared at a level may observe values of variables

Do the following programs leak information?

```
1. p := p + s
2. s := p
3. p := s; p := 1
4. if (s mod 2) = 0
     then p := 0 else p := 1
5. while s != 0 do { //nothing }
```

Leakage

- Suppose s is a secret variable and p is a public variable
- Subjects cleared at a level may observe values of variables

Do the following programs leak information?

```
1. p := p + s
2. s := p
3. p := s; p := 1
4. if (s mod 2) = 0
     then p := 0 else p := 1
5. while s != 0 do { //nothing }
```

Security condition

- **Noninterference** [Goguen and Meseguer 1982]: *actions of high-security users do not affect observations of low-security users*
- Intuition, as commonly adapted to programs: *changes to secret inputs do not cause observable change in public output*

VSI type system

Type system:

- set of rules for deriving facts about types of program expressions and commands
- e.g., $\Gamma \vdash \mathtt{c} : \tau\ \mathtt{cmd}$
  - Γ is a typing context: maps names of variables to their types
  - τ is a type: here will be H (high, secret) or L (low, public)
  - $\mathtt{c}$ is a command: assignment, if, while, etc.
  - $\Gamma \vdash \mathtt{c} : \tau\ \mathtt{cmd}$ means, in part, that $\mathtt{c}$ is a well-typed command

VSI type system

Theorem.

If $\Gamma \vdash \mathtt{c} : \tau\ \mathtt{cmd}$ *then* $\mathtt{c}$ *satisfies noninterference.*

Next lecture: the typing rules...

Upcoming events

- [today] Office hours canceled
- [May 8] A6 due
- [May 16] Final exam

Suspense is achieved by information control:
What you know. What the reader knows.
What the characters know.
– Tom Clancy