Web Security Part 4: Advanced Defenses

Summer University 2014, Prof. Tom Austin

Secure By Architecture

Developers are human: they will make mistakes.

How can we design tools so that the systems they create are inherently secure?

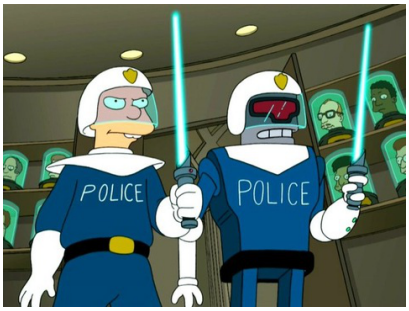
Success story: memory-safe languages

- Once upon a time, buffer overflow vulnerabilities were ubiquitous.
- Memory-safe languages manage memory automatically
  - Developer focus on functionality
  - Security-critical bugs are eliminated
- Buffer overflows have virtually disappeared
  - Except in your OS, web browser, etc.

Three Security Mechanisms

- **Object capabilities**: restrict what code can access.
- **Taint analysis**: protect critical fields from "dirty" data.
- **Information flow analysis**: Prevent secrets from leaking.

Object Capabilities: Restricting Access to Resources



The object-capability model borrows from capabilities in operating systems.

We'll review capabilities (C-lists) first.

Lampson's Access Control Matrix

- **Subjects** (users) index the rows

	OS	Accounting program	Accounting data	Insurance data	Payroll data
Bob	rx	rx	r	---	---
Alice	rx	rx	r	rw	rw
Sam	rwx	rwx	r	rw	rw
Accounting program	rx	rx	rw	rw	rw

Lampson's Access Control Matrix

- **Subjects** (users) index the rows
- **Objects** (resources) index the columns

	OS	Accounting program	Accounting data	Insurance data	Payroll data
Bob	rx	rx	r	---	---
Alice	rx	rx	r	rw	rw
Sam	rwx	rwx	r	rw	rw
Accounting program	rx	rx	rw	rw	rw

Are You Allowed to Do That?

- **Access control matrix** has **all** relevant info
- Could be 1000's of users, 1000's of resources
- Then matrix with 1,000,000's of entries
- How to manage such a large matrix?
- Need to check this matrix before access to any resource is allowed
- How to make this efficient?

Access Control Lists (ACLs)

- ACL: store access control matrix by **column**
- Example: ACL for **insurance data** is in **blue**

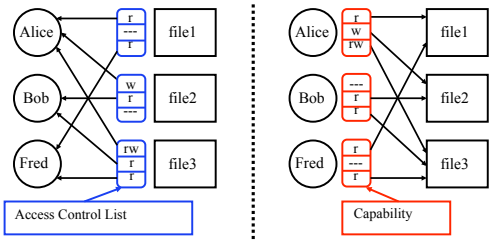
	OS	Accounting program	Accounting data	Insurance data	Payroll data
Bob	rx	rx	r	---	---
Alice	rx	rx	r	rw	rw
Sam	rwx	rwx	r	rw	rw
Accounting program	rx	rx	rw	rw	rw

Capabilities (or C-Lists)

- Store access control matrix by **row**
- Example: Capability for **Alice** is in **red**

	OS	Accounting program	Accounting data	Insurance data	Payroll data
Bob	rx	rx	r	---	---
Alice	rx	rx	r	rw	rw
Sam	rwx	rwx	r	rw	rw
Accounting program	rx	rx	rw	rw	rw

ACLs vs Capabilities



- Note that arrows point in opposite directions…
- With ACLs, still need to associate users to files

ACLs and capabilities seem equivalent, but there are some surprising differences.

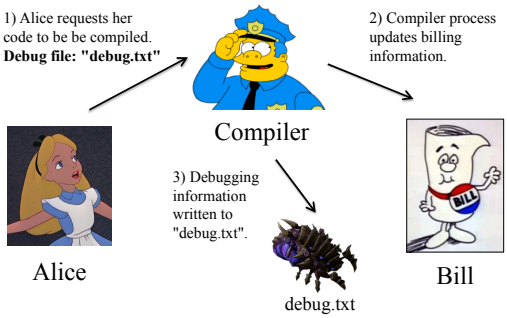
Let's highlight one of the limitations of ACLs.

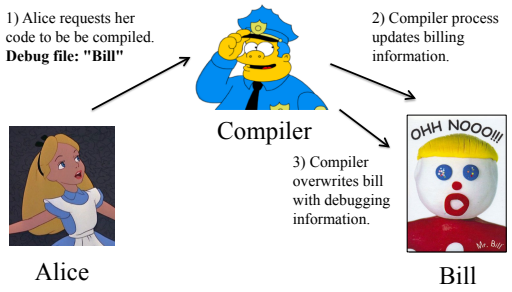
The Confused Deputy Problem

- Alice wants to compile her code on a 3rd party pay-per-use service.
  - Alice is able to specify a debug file
  - She is not allowed to overwrite the billing file
- The compiler process compiles the code and updates a billing file noting how much system resources she used.
- Simple, right! What could go wrong?

The access control matrix

	Compiler	BILL
Alice	x	---
Compiler	rx	rw





The compiler is a *deputy* acting on Alice's behalf, but he has *confused* his authority with hers.

Principle of least authority

We want to be suspicious of different processes and give each just the authority it needs to do its job.

Designators & Authorizations

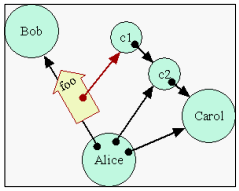
- With capabilities, we don't need a separate file system: a reference to a file is an intrinsic part of the design.
- How does this help with the confused deputy?
  1. Alice will not have a reference to Bill.
  2. The Compiler can use its authority for the billing task, but Alice's for the compilation.

Key analogy

A capability can roughly be thought of as a key:

- You can give a capability to another process, thus granting it your permissions
- It cannot easily be forged
- It cannot easily be stolen

Problems with the key analogy



- With careful design, capabilities (unlike keys) can be revoked.
- Capabilities can be composed, breaking the subj/obj distinction.
- For more details, see http://www.erights.org/elib/capability/duals/myths.html

Object Capability Model

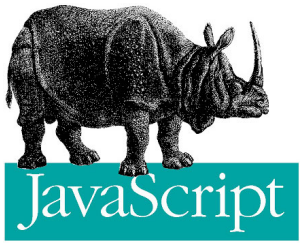
Object capabilities extend the concepts of capabilities to object-oriented programming languages.

In this system, capabilities are *unforgeable references*.

Required properties for object capabilities

1. Memory safety: Can't get a reference unless:
   - You created the object
   - You were given a reference to the object
2. Encapsulation
   - Provides a form of access control: can't access internals of an object.
3. Only references enable effects
4. No powerful references by default

Safely including JavaScript



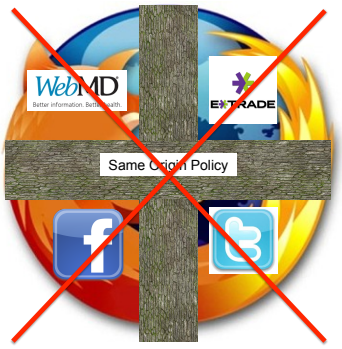
Motivations for using 3rd party JS

- Advertising
  - Historically, has been the source of many security problems.
- Social networks (FaceBook, etc)
- Maps
- Analytics
- Other useful widgets

The same origin policy

- The same origin policy states that only pages/ frames from the same origin can share properties and methods.
  - Origin = domain + protocol + port
- However, there are no restrictions based on where the source code is located, only on where it is run.

Same Origin Policy Prevents Mashups



Interferes with mashups

JavaScript Security: a tale of sadness and woe

ADVERTISING

Times Web Ads Show Security Breach

By ASHLEE VANCE
Published: September 14, 2009

OVER the weekend, some visitors to the Web site of The New York Times received a nasty surprise. An unknown person or group sneaked a rogue advertisement onto the site's pages.

Enlarge This Image

The malicious ad took over the browsers of many people visiting the site, as their screens filled with an image that seemed to show a scan for computer viruses. The visitors were then told that they needed to buy antivirus software to fix a problem, but the software was more snake oil than a useful program.

The fake antivirus scan on the Web site of The New York Times last weekend.

TWITTER
LINKEDIN
SIGN IN TO E-MAIL
PRINT
REPRINTS
SHARE

Sandboxing JavaScript

- `postMessage` (HTML5)
- Facebook JavaScript
- Yahoo's ADsafe
- Google's Caja

Caja details

- The name:
  - **CA**pabilities **JA**vaScript
  - Also, Spanish for 'box'
- Code is written in separate elements within the page.
- Code is rewritten through the use of a *cajoler*.
  - Any references to global properties are removed, unless specifically imported.

Simple Example

```
<div id="guest"></div>

<script type="text/javascript">
  caja.initialize({
  cajaServer: 'https://
caja.appspot.com/'});

caja.load(document.getElementById('guest'),
  undefined, function(frame) {
    frame.code('https://ab.com/guest.html',
               'text/html').run();
  });
</script>
```

The div element that will be the 'box' for the guest code

Simple Example

```
<div id="guest"></div>

<script type="text/javascript">
  caja.initialize({
  cajaServer: 'https://
caja.appspot.com/'});

caja.load(document.getElementById('guest'),
  undefined, function(frame) {
    frame.code('https://ab.com/guest.html',
               'text/html').run();
  });
</script>
```

Code is loaded from the specified location.

Problem 4.1: Caja Playground

Today we will look into Caja to see the translation process at work.
http://caja.appspot.com/

For details, see problem 1 in
http://cs31.cs.sjsu.edu/labs/exercise4.pdf.

Taint Analysis: Protecting against dirty data

Taint Analysis

- Any variable under the control of an outside user poses a security risk
  - Examples: SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), etc.
- Taint tracking tracks untrusted variables and prevents then from being used in unsafe operations

Taint Tracking History

- 1989 – Perl 3 included support for a taint mode
- 1996 – Netscape included support for a taint mode in server-side JavaScript
  - Also available in the client, but disabled by default
  - Later abandoned
- Ruby later implemented a taint mode; we'll review in more depth.

Ruby Crash Course

Hello World in Ruby

```
puts 'Hello world!'
```

```
class Person
  def initialize name # Constructor
    @name = name
  end

  def name            # Getter
    return @name
  end

  def name= newName   # Setter
    @name = newName
  end

  def say_hi          # Method
    puts "Hello, my name is #{@name}."
  end
end
```

Generating getters and setters

```
class Person
  attr_accessor :name
  def initialize name # Constructor
    @name = name
  end
  def say_hi          # Method
    puts "Hello, my name is #{@name}."
  end
end
```

Using a class in Ruby

```
p = Person.new "Joe"
puts "Name is #{p.name}"
p.say_hi
```

Regular Expressions in Ruby

```
s = "Hi, I'm Larry; this is my" +
    " brother Darryl, and this" +
    " is my other brother Darryl."
s.sub(/Larry/,'Laurent')
puts s
s.sub!(/Larry/,'Laurent')
puts s
puts s.sub(/brother/, 'frère')
puts s.gsub(/brother/, 'frère')
```

Problem 4.2 – Eliza in Ruby

See problem 2 in http://cs31.cs.sjsu.edu/labs/exercise4.pdf.

Taint Mode in Ruby

- Focused on protecting against integrity attacks.
  - E.g. Data pulled from an HTML form cannot be passed to `eval`.
- Does not handle implicit flows.
- Not possible to taint booleans or ints.
- Multiple ways to run in safe mode:
  - Use `-T` command line flag.
  - Include `$SAFE` variable in code.

`$SAFE` levels in Ruby

- 0 – No checking (default)
- 1
  - Tainted data cannot be passed to eval
  - Cannot load/require new files
- 2 – Can't change, make, or remove directories
- 3
  - New strings/objects are automatically tainted
  - Cannot untaint tainted values
- 4 – Safe objects become immutable

```
s = "puts 4-3".taint
$SAFE = 1 # Can't eval tainted data
s.untaint # Removes taint from data
puts s.tainted?
eval s

$SAFE = 3
s2 = "puts 2 * 7" # Tainted
s2.untaint # Won't work now
eval s2
eval s # this is OK
```

Problem 4.3 – Ruby Taint Tracking

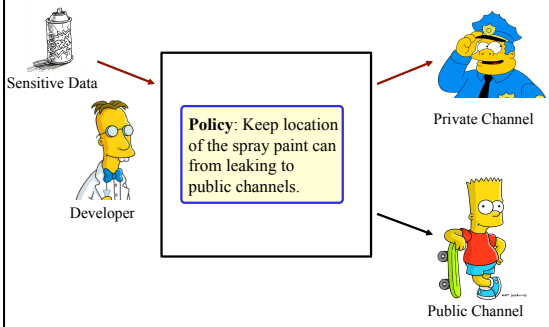
See problem 3 in http://cs31.cs.sjsu.edu/labs/exercise4.pdf.

Information Flow Analysis

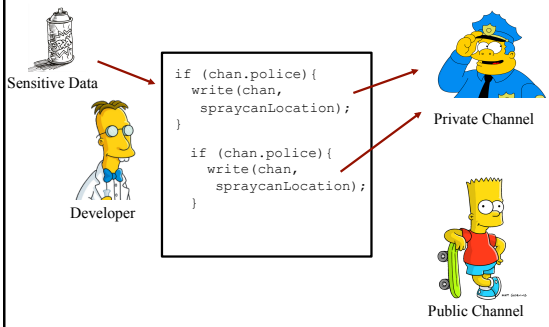
Taint & Information Flow Analysis

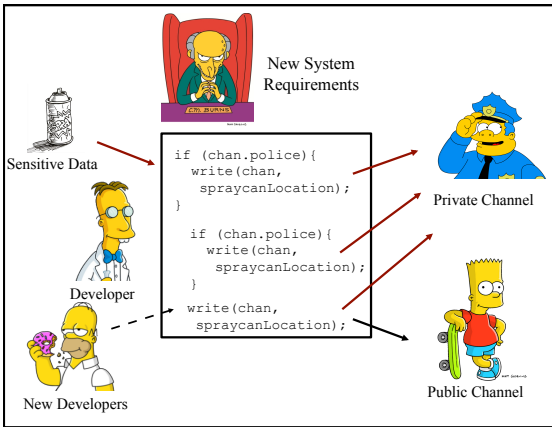
- Track/prevent flow of sensitive information.
- **Taint analysis** focuses on *integrity:* does "dirty" data corrupt trusted data?
  - Generally simpler to handle – usually ignores certain (impractical?) attacks.
- **Information flow analysis** handles integrity as well, but focuses on *confidentiality:* does secret data leak to public channels?

Challenge of Enforcing Security Policies



Challenge of Enforcing Security Policies





Policy Was Not Enforced

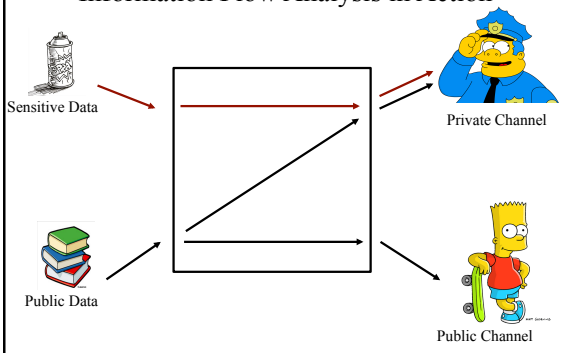
Additional Information Flow Challenges

Applications often make use of 3rd party libraries of questionable quality...

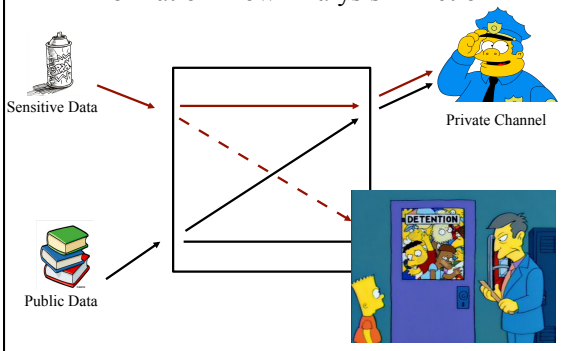
…or have vulnerabilities to code injection attacks...

…so we must assume that the attacker is able to inject code into our system.

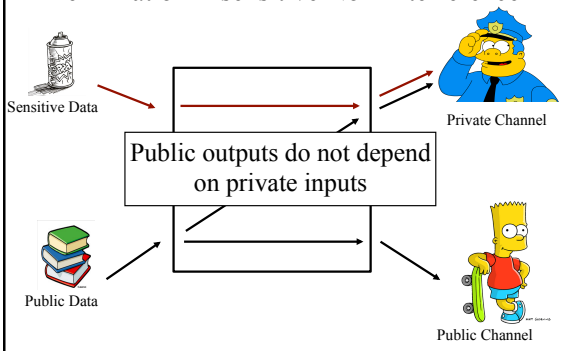
Information Flow Analysis in Action



Information Flow Analysis in Action



Termination-Insensitive Non-Interference



Explicit and Implicit Flows

```
spraycanLocation = "Kwik-E-Mart"police;
```

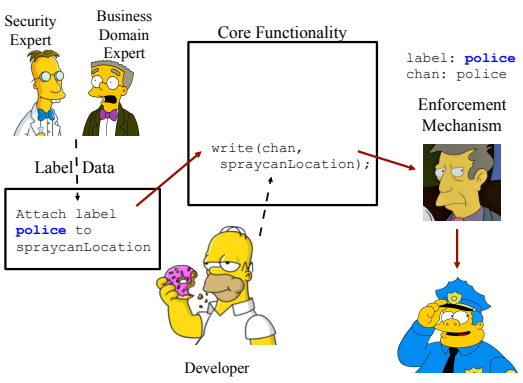
Location is only visible to the police.

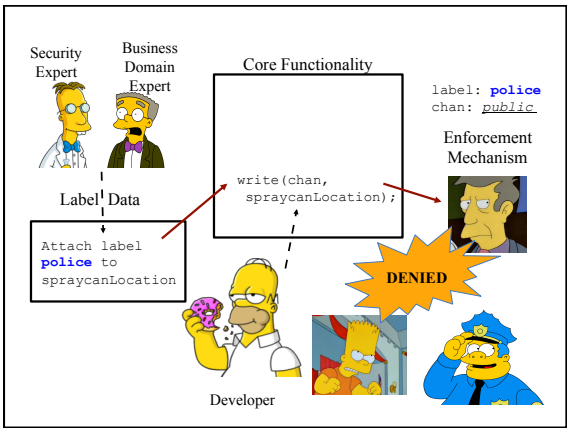
```
x = spraycanLocation;
```

Explicit flow from `spraycanLocation` to `x`.

```
if (x.charAt(0) < 'N') {
    firstCharMax = 12;
}
```

Implicit flow from `x` to `firstCharMax`.





Denning-style Static Analysis

- Done through a static certification process, perhaps integrated into a compiler.
- Data can flow down the lattice
- Programs can be guaranteed to be secure before the program is ever executed.

Static Analysis Certification

```
var secret = true^bank;
var y = true;
if (secret)
  y = false;
var leak = true;
if (y)
   leak = false;
```

NOT CERTIFIED

- Analysis ensures that private data does not affect public data.
- In this example, y's final value depends on x.
- [Denning 1976]

Purely Dynamic Info Flow Controls

- Instrument interpreter with runtime controls
- Implicit flows can be handled by:
  - Ignoring unsafe updates
  - Crashing on unsafe updates
  - Leaking some data (not satisfying noninterference)

A Tainting Approach

One obvious strategy: if a public variable is updated in a private context, make it private as well.

```
var secret = true^bank;
var y = true;
if (secret)
    y = false;
```

Set y=false^bank

Problem 4.4 – Exploring Tricky Code

The tainting approach will not work. Why not? Evaluate the following code using the tainting approach and give the results for when secret=true^bank and for secret=false^bank. Pay careful attention to the labels on the values.

```
var y = true;
var leak = true;
if (secret)y = false;
if (y) leak = false;
print(leak);
```

Dynamic Monitors Reject Executions

```
var secret = true;
var y = true;
if (secret)
  y = false;
```

Execution terminates to protect the value of `secret`.

Zdancewic 2002

Secure Multi-Execution

Executes program multiple:

- High execution
  - Sees all information
  - Only writes to authorized channels
- Low execution
  - Only sees public data
  - Writes to public channels
  - Confidential data replaced with default values.

Original Program

```
var pass = mkSecret("scytale");
if (pass[0]==("s"))
  write(chan,"s");
```

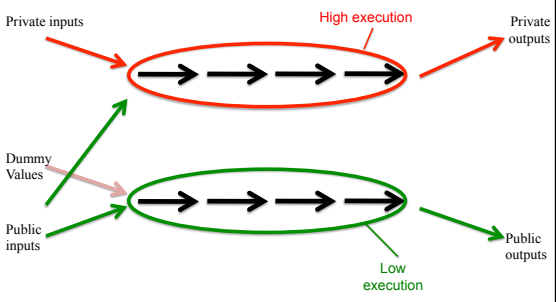
Low Execution Program

```
var pass="";
if (pass[0]
     ==("s"))
  write(chan,
     "s");
```

High Execution Program

```
var pass="scytale";
if (pass[0]
     ==("s")){
  write(chan,
     "s");
```

Secure Multi-Execution



Remaining Time

- You have the remaining period to work on your assignments
- Email your answers + zip file of source code to thomas.austin@sjsu.edu
  - Include your group name
  - Include **all** team member names *and email addresses*
- Due date: July 31, 2014