Lecture 7: Information flow control

Eran Tromer
Information Flow Control

- An information flow policy describes authorized paths along which information can flow.
- For example, Bell-La Padula describes a lattice-based information flow policy.
Example Lattice

(“Top Secret”, {“Soviet Union”, “East Germany”}),

(“Top Secret”, {“Soviet Union”})

(“Secret”, {“Soviet Union”, “East Germany”})

(“Secret”, {“Soviet Union”})

(“Secret”, {“East Germany”})

(“Unclassified”, ∅)
Lattices

- Dominance relationship $\geq$ defined in military security model is transitive and antisymmetric
- Therefore, it defines a lattice
- For two levels $a$ and $b$, neither $a \geq b$ nor $b \geq a$ might hold
- However, for every $a$ and $b$, there is a lowest upper bound $u$ for which $u \geq a$ and $u \geq b$, and a greatest lower bound $l$ for which $a \geq l$ and $b \geq l$
- There are also two elements $U$ and $L$ that dominate/are dominated by all levels
  - In example,
    $U = (\text{“Top Secret”}, \{\text{“Soviet Union”, “East Germany”}\})$
    $L = (\text{“Unclassified”}, \emptyset)$
Information Flow Control

- Input and output variables of program each have a (lattice-based) security classification $S()$ associated with them.

- For each program statement, compiler verifies whether information flow is secure.

- For example, $x = y + z$ is secure only if $S(x) \geq \text{lub}(S(y), S(z))$, where lub() is lowest upper bound.

- Program is secure if each of its statements is secure.
Implicit information flow

Conditional branches carry information about the condition:

```java
secret bool a;
public bool b;
...

b = a;  // Incorrect
if (a) {
    b = 0;
} else {
    b = 1;
}
```

Possible solution: assign label to program counter and include it in every operation. (Too conservative!)
Issues

- Label creep
  - Example: false flows
  - Example: branches

- Declassification
  - example: encryption
Implementation approaches

• Language-based IFC (JIF & others)
  – Compile-time tracking for Java, Haskell
  – Static analysis
  – Provable security
  – Label creep (false positives)

• Dynamic analysis
  – Language / bytecode / machine code
  – Tainting
  – False positives, false negatives
  – Hybrid solutions

• OS-based DIFC (Asbestos, HiStar, Flume)
  – Run-time tracking enforced by a trusted kernel
    • Works with any language, compiled or scripting, closed or open
Distributed Information Flow Control

1. Data tracking
2. Isolated declassification
Information Flow Control (IFC)

• Goal: track which secrets a process has seen
• Mechanism: each process gets a secrecy label
  – Label summarizes which categories of data a process is assumed to have seen
  – Examples:
    • { “Financial Reports” }
    • { “HR Documents” }
    • { “Financial Reports” and “HR Documents” }
Tags + Labels

Process $p$

$S_p = \{\text{Finance}, \text{HR}\}$

$D_p = \{\text{HR}\}$

Universe of Tags:

- Finance
- SecretProjects
- HR

change_label(\{\text{Finance}\});

tag_get(create_tag());

change_label(\{\text{Finance}, \text{HR}\});

change_label(\{\text{Finance}\});

change_label(\{});

DIFC: Declassification in action.

Any process can add any tag to its label.

DIFC Rule: A process can create a new tag; gets ability to declassify it.
Alice’s Data

$p$

$S_p = \{a\}$

Alice’s Data

$q$

$S_q = \{a\}$

declassifier

$S_d = \{a\}$

$D_d = \{a\}$

KERNEL
Communication Rule

\[ S_p = \{ \text{HR} \} \quad \quad S_q = \{ \text{HR, Finance} \} \]

\( p \) can send to \( q \) iff \( S_p \subseteq S_q \)
Question:

• What interface should the kernel expose to applications?
  – Easier question than “is my OS implementation secure”
  – Easy to get it wrong!
Approach 1: “Floating Labels”
[IX, Asbestos]

\[ S_p = \{a\} \quad S_q = \{a\} \]
Floaters Leaks Data

Alice’s Data

\( S = \{a\} \)

1001

attacker

\( S = \{a\} \)

Leak file

\( S = \{\} \)

\( S = \{\} \)

\( S = \{\} \)

\( S = \{\} \)

\( S = \{\} \)

\( S = \{\} \)

\( S = \{\} \)

\( S = \{\} \)
Approach 2: “Set Your Own”

[HiStar/Flume]

\[ S_p = \{a\} \]
\[ O_p = \{a^-, a^+\} \]
\[ S_q = \{a\} \]
\[ O_q = \{a^+\} \]

Rule: \( S_p \subseteq S_q \) necessary precondition for send
Case study: Flume

• Goal: User-level implementation
  – apt-get install flume

• Approach:
  – System Call Delegation [*Ostia* by Garfinkel et al, 2003]
  – Use Linux 2.6 (or OpenBSD 3.9)
System Call Delegation

open("/hr/LayoffPlans", O_RDONLY);

Linux Kernel

Web App

glibc

Layoff Plans
System Call Delegation

open("/hr/LayoffPlans", O_RDONLY);
System Calls

- IPC: read, write, select
- Process management: fork, exit, getpid
- DIFC: create tag, change label, fetch label, send capabilities, receive capabilities
How To Prove Security

• Property: Noninterference
Noninterference [Goguen & Meseguer ’82]

Experiment #1:

\[ p \]

\[ S_p = \{ a \} \]

\[ p' \]

\[ S_{p'} = \{ a \} \]

Experiment #2:

\[ q \]

\[ S_q = \{ \} \]

"HIGH"

\[ \parallel \]

"LOW"

\[ S_q = \{ \} \]
How To Prove Security (cont.)

• Property: Noninterference
• Process Algebra model for a DIFC OS
  – Communicating Sequential Processes (CSP)
• Proof that the model fits the definition
Proving noninterference

- Formalism (Communicating Sequential Formalisms)
- Consider any possible system (with arbitrary user applications)
- Induction over all possible sequences of moves a system can make (i.e., traces)
- At each step, case-by-case analysis of all system calls in the interface.
  - Prove no “interference”
Results on Flume

• Allows adoption of Unix software?
  – 1,000 LOC launcher/declassifier
  – 1,000 out of 100,000 LOC in MoinMoin changed
  – Python interpreter, Apache, unchanged

• Solves security vulnerabilities?
  – Without our knowing, we inherited two ACL bypass bugs from MoinMoin
  – Both are not exploitable in Flume’s MoinMoin

• Performance?
  – Performs within a factor of 2 of the original on read and write benchmarks

• Example App: MoinMoin Wiki
Flume Limitations

• Bigger TCB than HiStar / Asbestos
  – Linux stack (Kernel + glibc + linker)
  – Reference monitor (~22 kLOC)
• Covert channels via disk quotas
• Confined processes like MoinMoin don’t get full POSIX API.
  – spawn() instead of fork() & exec()
  – flume_pipe() instead of pipe()
## DIFC vs. Traditional MAC

<table>
<thead>
<tr>
<th>Traditional MAC</th>
<th>DIFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Military Systems</td>
<td>Web-based Systems, Desktops</td>
</tr>
<tr>
<td>Superior Contraining Users</td>
<td>Developers Protecting Themselves from Bugs</td>
</tr>
<tr>
<td>Centralized Declassification Policy</td>
<td>Application-Specific Declassification Policies</td>
</tr>
</tbody>
</table>
What are we trusting?

• Proof of noninterference
• Platform
  – Hardware
  – Virtual machine manager
  – Kernel
• Reference monitor
• Declassifier
  – Human decisions
• Covert channels
• Physical access
• User authentication
Practical barriers to deployment of Information Flow Control systems

• Programming model confuses programmers
• Lack of support:
  – Applications
  – Libraries
  – Operating systems
• People don’t care about security until it's too late
Quantitative information flow control

• Quantify how much information (#bits) is leaking from (secret) inputs to outputs

• Approach: dynamic analysis (extended tainting) followed by flow graph analysis

• Example: battleship online game