Lecture 20

- Compiler-based mechanisms
- Execution-based mechanisms
- The confinement problem
- Isolation: virtual machines, sandboxes
- Covert channels
  - Detection
  - Mitigation
Procedure Calls

\[ \text{tm}(a, b); \]

From previous slides, to be secure, \( \text{lub}(x, i) \leq y \) must hold

- In call, \( x \) corresponds to \( a \), \( y \) to \( b \)
- Means that \( \text{lub}(a, i) \leq b \), or \( a \leq b \)

More generally:

\[ \text{proc } pn(i_1, \ldots, i_m: \text{int}; \text{var } o_1, \ldots, o_n: \text{int}) \]
\[ \text{begin } S \text{ end; } \]

- \( S \) must be secure
- For all \( j \) and \( k \), if \( i_j \leq o_k \), then \( x_j \leq y_k \)
- For all \( j \) and \( k \), if \( o_j \leq o_k \), then \( y_j \leq y_k \)
Exceptions

proc copy(x: int class { x };
            var y: int class Low)
var sum: int class { x };
      z: int class Low;
begin
    y := z := sum := 0;
    while z = 0 do begin
      sum := sum + x;
      y := y + 1;
    end
end
Exceptions (cont)

- When sum overflows, integer overflow trap
  - Procedure exits
  - Value of $x$ is $\text{MAXINT}/y$
  - Info flows from $y$ to $x$, but $x \leq y$ never checked

- Need to handle exceptions explicitly
  - Idea: on integer overflow, terminate loop
    
    ```
    on integer_overflow_exception sum do z := 1;
    ```
  - Now info flows from $sum$ to $z$, meaning $\text{sum} \leq z$
  - This is false ($\text{sum} = \{ x \}$ dominates $z = \text{Low}$)
Infinite Loops

```
proc copy(x: int 0..1 class { x });
    var y: int 0..1 class Low)
begin
    y := 0;
    while x = 0 do
        (* nothing *);
    y := 1;
end
```

- If $x = 0$ initially, infinite loop
- If $x = 1$ initially, terminates with $y$ set to 1
- No explicit flows, but implicit flow from $x$ to $y$
Semaphores

Use these constructs:

\begin{align*}
\text{wait}(x) & : \quad \text{if } x = 0 \text{ then block until } x > 0; \quad x := x - 1; \\
\text{signal}(x) & : \quad x := x + 1;
\end{align*}

- $x$ is semaphore, a shared variable
- Both executed atomically

Consider statement

\begin{align*}
\text{wait}(\text{sem}); & \quad x := x + 1;
\end{align*}

- Implicit flow from $\text{sem}$ to $x$
  - Certification must take this into account!
Flow Requirements

• Semaphores in *signal* irrelevant
  – Don’t affect information flow in that process

• Statement $S$ is a wait
  – $\text{shared}(S)$: set of shared variables read
    • Idea: information flows out of variables in shared($S$)
  – $\text{fglb}(S)$: glb of assignment targets following $S$
  – So, requirement is $\text{shared}(S) \leq \text{fglb}(S)$

• begin $S_1; \ldots S_n$ end
  – All $S_i$ must be secure
  – For all $i$, $\text{shared}(S_i) \leq \text{fglb}(S_i)$
Example

begin
  \( x := y + z; \) \hspace{1cm} (\ast S_1 \ast)
  \text{wait}(\text{sem}); \hspace{1cm} (\ast S_2 \ast)
  \( a := b \times c - x; \) \hspace{1cm} (\ast S_3 \ast)
end

\begin{itemize}
  \item Requirements:
    \begin{itemize}
      \item \( \text{lub}(y, z) \leq x \)
      \item \( \text{lub}(b, c, x) \leq a \)
      \item \( \text{sem} \leq a \)
      \end{itemize}
    \item Because \( \text{fglb}(S_2) = a \) and \( \text{shared}(S_2) = \text{sem} \)
\end{itemize}
Concurrent Loops

- Similar, but wait in loop affects all statements in loop
  - Because if flow of control loops, statements in loop before wait may be executed after wait

- Requirements
  - Loop terminates
  - All statements $S_1, \ldots, S_n$ in loop secure
  - $lub(\text{shared}(S_1), \ldots, \text{shared}(S_n)) \leq glb(t_1, \ldots, t_m)$
    - Where $t_1, \ldots, t_m$ are variables assigned to in loop
Loop Example

while $i < n$ do begin
  $a[i] := item$; (* $S_1$ *)
  wait($sem$); (* $S_2$ *)
  $i := i + 1$; (* $S_3$ *)
end

• Conditions for this to be secure:
  – Loop terminates, so this condition met
  – $S_1$ secure if $lub(i, item) \leq a[i]$
  – $S_2$ secure if $sem \leq i$ and $sem \leq a[i]$
  – $S_3$ trivially secure

May 15, 2013
ECS 235B Spring Quarter 2013
cobegin/coend

cobegin
  \( x := y + z; \) \hspace{1cm} (* S_1 *)
  \( a := b \times c - y; \) \hspace{1cm} (* S_2 *)
coend

- No information flow among statements
  - For \( S_1 \), \( \text{lub}(y, z) \leq x \)
  - For \( S_2 \), \( \text{lub}(b, c, y) \leq a \)

- Security requirement is both must hold
  - So this is secure if \( \text{lub}(y, z) \leq x \land \text{lub}(b, c, y) \leq a \)
Soundness

- Above exposition intuitive
- Can be made rigorous:
  - Express flows as types
  - Equate certification to correct use of types
  - Checking for valid information flows same as checking types conform to semantics imposed by security policy
Execution-Based Mechanisms

- Detect and stop flows of information that violate policy
  - Done at run time, not compile time
- Obvious approach: check explicit flows
  - Problem: assume for security, $x \leq y$
    
    ```
    if x = 1 then y := a;
    ```
  - When $x \neq 1$, $x = \text{High}$, $y = \text{Low}$, $a = \text{Low}$, appears okay
    — but implicit flow violates condition!
Fenton’s Data Mark Machine

- Each variable has an associated class
- Program counter (PC) has one too
- Idea: branches are assignments to PC, so you can treat implicit flows as explicit flows
- Stack-based machine, so everything done in terms of pushing onto and popping from a program stack
Instruction Description

- *skip* means instruction not executed
- *push(x, x)* means push variable $x$ and its security class $x$ onto program stack
- *pop(x, x)* means pop top value and security class from program stack, assign them to variable $x$ and its security class $x$ respectively
Instructions

• $x := x + 1$ (increment)
  - Same as:
    ```
    if $PC \leq x$ then $x := x + 1$ else skip
    ```

• if $x = 0$ then goto $n$ else $x := x - 1$ (branch and save $PC$ on stack)
  - Same as:
    ```
    if $x = 0$ then begin
      push($PC$, $PC$); $PC := \text{lub}\{PC, x\}$; $PC := n$;
      end else if $PC \leq x$ then
        $x := x - 1$
      else
        skip;
    ```
More Instructions

- if’ $x = 0$ then goto $n$ else $x := x - 1$
  (branch without saving PC on stack)
  - Same as:
    - if $x = 0$ then
      - if $x \leq PC$ then $PC := n$ else $skip$
    else
      - if $PC \leq x$ then $x := x - 1$ else $skip$
More Instructions

• return (go to just after last if)
  – Same as:
    \[ \text{pop}(PC, PC); \]

• halt (stop)
  – Same as:
    \[ \text{if program stack empty then halt} \]
  – Note stack empty to prevent user obtaining information from it after halting
Example Program

1  if $x = 0$ then goto 4 else $x := x - 1$
2  if $z = 0$ then goto 6 else $z := z - 1$
3  halt
4  $z := z + 1$
5  return
6  $y := y + 1$
7  return

• Initially $x = 0$ or $x = 1$, $y = 0$, $z = 0$
• Program copies value of $x$ to $y$
Example Execution

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>PC</th>
<th>PC</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Low</td>
<td>—</td>
<td>Low ≤ x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>z</td>
<td>(3, Low)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>z</td>
<td>(3, Low)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>
Handling Errors

• Ignore statement that causes error, but continue execution
  – If aborted or a visible exception taken, user could deduce information
  – Means errors cannot be reported unless user has clearance at least equal to that of the information causing the error
Variable Classes

• Up to now, classes fixed
  – Check relationships on assignment, etc.
• Consider variable classes
  – Fenton’s Data Mark Machine does this for $PC$
  – On assignment of form $y := f(x_1, \ldots, x_n)$, $y$
    changed to $lub(x_1, \ldots, x_n)$
  – Need to consider implicit flows, also
Example Program

(* Copy value from x to y
* Initially, x is 0 or 1 *)
proc copy(x: int class { x };
    var y: int class { y })
var z: int class variable { Low };
begin
    y := 0;
    z := 0;
    if x = 0 then z := 1;
    if z = 0 then y := 1;
end;

• z changes when z assigned to
• Assume y < x
Analysis of Example

- $x = 0$
  - $z := 0$ sets $\bar{z}$ to Low
  - if $x = 0$ then $z := 1$ sets $z$ to 1 and $\bar{z}$ to $x$
  - So on exit, $y = 0$

- $x = 1$
  - $z := 0$ sets $\bar{z}$ to Low
  - if $z = 0$ then $y := 1$ sets $y$ to 1 and checks that $\text{lub}\{\text{Low}, \bar{z}\} \leq y$
  - So on exit, $y = 1$

- Information flowed from $x$ to $y$ even though $y < x$
Handling This (1)

• Fenton’s Data Mark Machine detects implicit flows violating certification rules
Handling This (2)

- Raise class of variables assigned to in conditionals even when branch not taken
- Also, verify information flow requirements even when branch not taken
- Example:
  - In `if x = 0 then z := 1`, `z` raised to `x` whether or not `x = 0`
  - Certification check in next statement, that `z ≤ y`, fails, as `z = x` from previous statement, and `y ≤ x`
Handling This (3)

- Change classes only when explicit flows occur, but *all* flows (implicit as well as explicit) force certification checks

- Example
  - When \( x = 0 \), first “if” sets \( z \) to Low then checks \( x \leq z \)
  - When \( x = 1 \), first “if” checks that \( x \leq z \)
  - This holds if and only if \( x = \text{Low} \)
    - Not possible as \( y < x = \text{Low} \) and there is no such class
Examples

• Use access controls of various types to inhibit information flows

• Security Pipeline Interface
  – Analyzes data moving from host to destination

• Secure Network Server Mail Guard
  – Controls flow of data between networks that have different security classifications
Security Pipeline Interface

- SPI analyzes data going to, from host
  - No access to host main memory
  - Host has no control over SPI
Use

- Store files on first disk
- Store corresponding crypto checksums on second disk
- Host requests file from first disk
  - SPI retrieves file, computes crypto checksum
  - SPI retrieves file’s crypto checksum from second disk
  - If a match, file is fine and forwarded to host
  - If discrepancy, file is compromised and host notified
- Integrity information flow restricted here
  - Corrupt file can be seen but will not be trusted
Secure Network Server Mail Guard (SNSMG)

- Filters analyze outgoing messages
  - Check authorization of sender
  - Sanitize message if needed (words and viruses, etc.)

- Uses type checking to enforce this
  - Incoming, outgoing messages of different type
  - Only appropriate type can be moved in or out
Confinement

• What is the problem?
• Isolation: virtual machines, sandboxes
• Detecting covert channels
Example Problem

• Server balances bank accounts for clients
• Server security issues:
  – Record correctly who used it
  – Send *only* balancing info to client
• Client security issues:
  – Log use correctly
  – Do not save or retransmit data client sends
Generalization

- Client sends request, data to server
- Server performs some function on data
- Server returns result to client
- Access controls:
  - Server must ensure the resources it accesses on behalf of client include *only* resources client is authorized to access
  - Server must ensure it does not reveal client’s data to any entity not authorized to see the client’s data
Confinement Problem

• Problem of preventing a server from leaking information that the user of the service considers confidential
Total Isolation

• Process cannot communicate with any other process
• Process cannot be observed

Impossible for this process to leak information
  – Not practical as process uses observable resources such as CPU, secondary storage, networks, etc.
Example

- Processes $p$, $q$ not allowed to communicate
  - But they share a file system!
- Communications protocol:
  - $p$ sends a bit by creating a file called 0 or 1, then a second file called $send$
    - $p$ waits until $send$ is deleted before repeating to send another bit
  - $q$ waits until file $send$ exists, then looks for file 0 or 1; whichever exists is the bit
    - $q$ then deletes 0, 1, and $send$ and waits until $send$ is recreated before repeating to read another bit
Covert Channel

• A path of communication not designed to be used for communication

• In example, file system is a (storage) covert channel
Rule of Transitive Confinement

• If $p$ is confined to prevent leaking, and it invokes $q$, then $q$ must be similarly confined to prevent leaking

• Rule: if a confined process invokes a second process, the second process must be as confined as the first
Lipner’s Notes

• All processes can obtain rough idea of time
  – Read system clock or wall clock time
  – Determine number of instructions executed

• All processes can manipulate time
  – Wait some interval of wall clock time
  – Execute a set number of instructions, then block
Kocher’s Attack

- This computes $x = a^z \mod n$, where $z = z_0 \ldots z_{k-1}$

\[
x := 1; \text{atmp} := a;
\text{for } i := 0 \text{ to } k-1 \text{ do begin}
\quad \text{if } z_i = 1 \text{ then}
\qquad x := (x * \text{atmp}) \mod n;
\qquad \text{atmp} := (\text{atmp} * \text{atmp}) \mod n;
\quad \text{end}
\text{result} := x;
\]

- Length of run time related to number of 1 bits in $z$
Isolation

• Present process with environment that appears to be a computer running only those processes being isolated
  – Process cannot access underlying computer system, any process(es) or resource(s) not part of that environment
  – A virtual machine

• Run process in environment that analyzes actions to determine if they leak information
  – Alters the interface between process(es) and computer