Lecture 24
November 27, 2023

Exceptions

```
proc copy(x: integer class { x };
                    var y: integer class Low);
var sum: integer 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 *sum* is MAXINT/*y*
  - Information flows from *y* to *sum*, but <u>*sum*</u> ≤ <u>*y*</u> never checked
- Need to handle exceptions explicitly
  - Idea: on integer overflow, terminate loop

    on integer_overflow_exception *sum* **do** *z := 1;*
  - Now information flows from *sum* to *z*, meaning <u>*sum*</u> ≤ <u>*z*</u>
  - This is false (<u>*sum*</u> = { x } dominates <u>*z*</u> = Low)

Infinite Loops

```
proc copy(x: integer 0..1 class { x };
                 var y: integer 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:

wait(x): **if** $x = 0$ **then block until** $x > 0$; $x := x - 1$;

signal(x): $x := x + 1$;

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

Consider statement

$$\texttt{wait}(sem);\ x := x + 1;$$

- Implicit flow from *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*
  - shared(S): set of shared variables read
    - Idea: information flows out of variables in shared(S)
  - fglb(S): glb of assignment targets *following S*
  - So, requirement is shared(S) ≤ fglb(S)

- begin S_1; … S_n end
  - All S_i must be secure
  - For all i, shared(S_i) ≤ fglb(S_i)

Example

```
begin
    x := y + z;          (* S₁ *)
    wait(sem);           (* S₂ *)
    a := b * c - x;      (* S₃ *)
end
```

- Requirements:
  - lub{ $\underline{y}$, $\underline{z}$ } ≤ $\underline{x}$
  - lub{ $\underline{b}$, $\underline{c}$, $\underline{x}$ } ≤ $\underline{a}$
  - $\underline{sem}$ ≤ $\underline{a}$
    - Because fglb(S_2) = $\underline{a}$ and shared(S_2) = sem

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, …, S_n in loop secure
  - lub{ shared(S_1), …, shared(S_n) } $\leq$ glb(t_1, …, t_m)
    - Where t_1, …, t_m are variables assigned to in loop

Loop Example

```
while i < n do begin
    a[i] := item;      (* S₁ *)
    wait(sem);         (* S₂ *)
    i := i + 1;        (* S₃ *)
end
```

- Conditions for this to be secure:
  - Loop terminates, so this condition met
  - S_1 secure if lub{ i, _item_ } ≤ _a[i]_
  - S_2 secure if _sem_ ≤ _i_ and _sem_ ≤ _a[i]_
  - S_3 trivially secure

cobegin/coend

cobegin

$$x := y + z; \qquad (* \ S_1 \ *)$$

$$a := b * c - y; \qquad (* \ S_2 \ *)$$

coend

- No information flow among statements
  - For S_1, lub$\{ \underline{y}, \underline{z} \} \le \underline{x}$
  - For S_2, lub$\{ \underline{b}, \underline{c}, \underline{y} \} \le \underline{a}$

- Security requirement is both must hold
  - So this is secure if lub$\{ \underline{y}, \underline{z} \} \le \underline{x} \wedge$ lub$\{ \underline{b}, \underline{c}, \underline{y} \} \le \underline{a}$

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, $x \leq y$

$$\mathbf{if} \; x \; = \; 1 \; \mathbf{then} \; y \; := \; a;$$

  - When $x \neq 1$, x = High, y = Low, a = 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`: instruction not executed

- `push(x, x)`: push variable *x* and its security class *x* onto program stack

- `pop(x, x)` : 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 $\underline{PC} \leq \underline{x}$ **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$(PC, \underline{PC})$; $\underline{PC}$:= lub$\{\underline{PC}, x\}$; PC := n;
     end else if $\underline{PC} \leq \underline{x}$ **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 $\underline{x} \leq \underline{PC}$ **then** $PC := n$ **else** *skip*
    else
      if $\underline{PC} \leq \underline{x}$ **then** $x := x - 1$ **else** *skip*

More Instructions

- **`return`** (go to just after last *if*)
  - Same as:
    `pop(`*PC*`, `*PC*`);`
- **`halt`** (stop)
  - Same as:
    `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: Initial Setting

x	*y*	*z*	*PC*	*PC*	*stack*	*check*
1	0	0	1	Low	—	

Example Execution: Step 1

x	*y*	*z*	*PC*	*PC*	*stack*	*check*
1	0	0	1	Low	—	
0	0	0	2	Low	—	Low ≤ *x*

if x = 0 **then goto** 4 **else** x := x - 1

```
if x = 0 then begin
    push(PC, PC); PC := lub{PC, x}; PC := n;
end else if PC ≤ x then
    x := x - 1
else
    skip;
```

Example Execution: Step 2

x	*y*	*z*	*PC*	*PC̲*	*stack*	*check*
1	0	0	1	Low	—	
0	0	0	2	Low	—	Low ≤ *x̲*
0	0	0	6	*z̲*	(3, Low)	*PC̲* ≤ *y̲*

```
if z = 0 then goto 6 else z := z - 1
```

```
if z = 0 then begin
    push(PC, PC̲); PC̲ := lub{PC̲, z̲}; PC := n;
end else if PC̲ ≤ z̲ then
    z := z - 1
else
    skip;
```

Example Execution: Step 3

x	y	z	PC	<u>PC</u>	stack	check
1	0	0	1	Low	—	
0	0	0	2	Low	—	Low ≤ <u>x</u>
0	0	0	6	<u>z</u>	(3, Low)	<u>PC</u> ≤ <u>y</u>
0	1	0	7	<u>z</u>	(3, Low)	

y := y + 1

---

if <u>PC</u> ≤ <u>y</u> **then** y := y + 1 **else** *skip*

Example Execution: Step 4

x	y	z	PC	$\underline{PC}$	stack	check
1	0	0	1	Low	—	
0	0	0	2	Low	—	Low $\leq \underline{x}$
0	0	0	6	$\underline{z}$	(3, Low)	$\underline{PC} \leq \underline{y}$
0	1	0	7	$\underline{z}$	(3, Low)	

return

$$\textbf{pop}(PC, \ \underline{PC});$$

Example Execution: Step 5

x	y	z	PC	$\underline{PC}$	stack	check
1	0	0	1	Low	—	
0	0	0	2	Low	—	Low ≤ $\underline{x}$
0	0	0	6	$\underline{z}$	(3, Low)	$\underline{PC} \le \underline{y}$
0	1	0	7	$\underline{z}$	(3, Low)	
0	1	0	3	Low	—	

halt

> **if** *program stack empty* **then** *halt*

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

ECS 235A, Computer and Information Security

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 $y := f(x_1, …, x_n)$, $\underline{y}$ changed to lub{ $\underline{x}_1, …, \underline{x}_n$ }
  - Need to consider implicit flows, also

Example Program

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

- $\underline{z}$ changes when z assigned to
- Assume $\underline{y} < \underline{x}$ (that is, $\underline{x}$ strictly dominates $\underline{y}$; they are not equal)

Analysis of Example

- *x* = 0
  - `z := 0` sets *z̲* to Low
  - `if x = 0 then z := 1` sets *z* to 1 and *z̲* to *x̲*
  - So on exit, *y* = 0

- *x* = 1
  - `z := 0` sets *z̲* to Low
  - `if z = 0 then y := 1` sets *y* to 1 and checks that lub{Low, *z̲*} ≤ *y̲*
  - So on exit, *y* = 1

- Information flowed from *x̲* to *y̲* even though *y̲* < *x̲*

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`, $\underline{z}$ raised to $\underline{x}$ whether or not $x = 0$
  - Certification check in next statement, that $\underline{z} \leq \underline{y}$, fails, as $\underline{z} = \underline{x}$ from previous statement, and $\underline{y} < \underline{x}$

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 $\underline{z}$ to Low, then checks $\underline{x} \leq \underline{z}$
  - When x = 1, first **if** checks $\underline{x} \leq \underline{z}$
  - This holds if and only if $\underline{x}$ = Low
    - Not possible as $\underline{y} < \underline{x}$ = Low by assumption and there is no class that Low strictly dominates

Integrity Mechanisms

- The above also works with Biba, as it is mathematical dual of Bell-LaPadula

- All constraints are simply duals of confidentiality-based ones presented above

Example 1

For information flow of assignment statement:

$$y := f(x_1, \ldots, x_n)$$

the relation glb$\{ \underline{x}_1, \ldots, x_n \} \geq \underline{y}$ must hold

- Why? Because information flows from $x_1, \ldots, x_n$ to y, and under Biba, information must flow from a higher (or equal) class to a lower one

Example 2

For information flow of conditional statement:

$$\textbf{if } f(x_1, \ldots, x_n) \textbf{ then } S_1; \textbf{ else } S_2; \textbf{ end};$$

then the following must hold:

- S_1, S_2 must satisfy integrity constraints
- glb{ $\underline{x}_1$, …, $\underline{x}_n$ } ≥ lub{$\underline{y}$ | y target of assignment in S_1, S_2 }

Example Information Flow Control Systems

- Privacy and Android Cell Phones
  - Analyzes data being sent from the phone
- Firewalls

Privacy and Android Cell Phones

- Many commercial apps use advertising libraries to monitor clicks, fetch ads, display them
  - So they send information, ostensibly to help tailor advertising to you
- Many apps ask to have full access to phone, data
  - This is because of complexity of permission structure of Android system
- Ads displayed with privileges of app
  - And if they use Javascript, that executes with those privileges
  - So if it has full access privilege, it can send contact lists, other information to others
- Information flow problem as information is flowing from phone to external party

Analyzing Android Flows

- Android based on Linux
  - App executables in bytecode format (Dalvik executables, or DEX) and run in Dalvik VM
  - Apps event driven
  - Apps use system libraries to do many of their functions
  - Binder subsystem controls interprocess communication
- Analysis uses 2 security levels, *untainted* and *tainted*
  - No categories, and *tainted < untainted*

TaintDroid: Checking Information Flows

- All objects tagged *tainted* or *untainted*
  - Interpreters, Binder augmented to handle tags

- Android native libraries trusted
  - Those communicating externally are *taint sinks*

- When untrusted app invokes a taint sink library, taint tag of data is recorded

- Taint tags assigned to external variables, library return values
  - These are assigned based on knowledge of what native code does

- Files have single taint tag, updated when file is written

- Database queries retrieve information, so tag determined by database query responder

TaintDroid: Checking Information Flows

- Information from phone sensor may be sensitive; if so, *tainted*
  - TaintDroid determines this from characteristics of information

- Experiment 1 (2010): selected 30 popular apps out of a set of 358 that required permission to access Internet, phone location, camera, or microphone; also could access cell phone information
  - 105 network connections accessed *tainted* data
  - 2 sent phone identification information to a server
  - 9 sent device identifiers to third parties, and 2 didn't tell user
  - 15 sent location information to third parties, none told user
  - No false positives

TaintDroid: Checking Information Flows

- Experiment 2 (2012): revisited 18 out of the 30 apps (others did not run on current version of Android)
  - 3 still sent location information to third parties
  - 8 sent device identification information to third parties without consent
    - 3 of these did so in 2010 experiment
    - 5 were new
  - 2 new flows that could reveal *tainted* data
  - No false positives

Firewalls

- Host that mediates access to a network
  - Allows, disallows accesses based on configuration and type of access
- Example: block Conficker worm
  - Conficker connects to botnet, which can use system for many purposes
    - Spreads through a vulnerability in a particular network service
  - Firewall analyze packets using that service remotely, and look for Conficker and its variants
    - If found, packets discarded, and other actions may be taken
  - Conficker also generates list of host names, tried to contact botnets at those hosts
    - As set of domains known, firewall can also block outbound traffic to those hosts

Filtering Firewalls

- Access control based on attributes of packets and packet headers
  - Such as destination address, port numbers, options, etc.
  - Also called a *packet filtering firewall*
  - Does not control access based on content
  - Examples: routers, other infrastructure systems

Proxy

- Intermediate agent or server acting on behalf of endpoint without allowing a direct connection between the two endpoints
    - So each endpoint talks to proxy, thinking it is talking to other endpoint
    - Proxy decides whether to forward messages, and whether to alter them

Proxy Firewall

- Access control done with proxies
  - Usually bases access control on content as well as source, destination addresses, etc.
  - Also called an *applications level* or *application level firewall*
  - Example: virus checking in electronic mail
    - Incoming mail goes to proxy firewall
    - Proxy firewall receives mail, scans it
    - If no virus, mail forwarded to destination
    - If virus, mail rejected or disinfected before forwarding

Example

- Want to scan incoming email for malware

- Firewall acts as recipient, gets packets making up message and reassembles the message
  - It then scans the message for malware
  - If none, message forwarded
  - If some found, mail is discarded (or some other appropriate action)

- As email reassembled at firewall by a mail agent acting on behalf of mail agent at destination, it's a proxy firewall (application layer firewall)

Stateful Firewall

- Keeps track of the state of each connection

- Similar to a proxy firewall
    - No proxies involved, but this can examine contents of connections
    - Analyzes each packet, keeps track of state
    - When state indicates an attack, connection blocked or some other appropriate action taken

Network Organization: DMZ

- DMZ is portion of network separating a purely internal network from external network

- Usually put systems that need to connect to the Internet here

- Firewall separates DMZ from purely internal network

- Firewall controls what information is allowed to flow through it
  - Control is bidirectional; it control flow in both directions

One Setup of DMZ
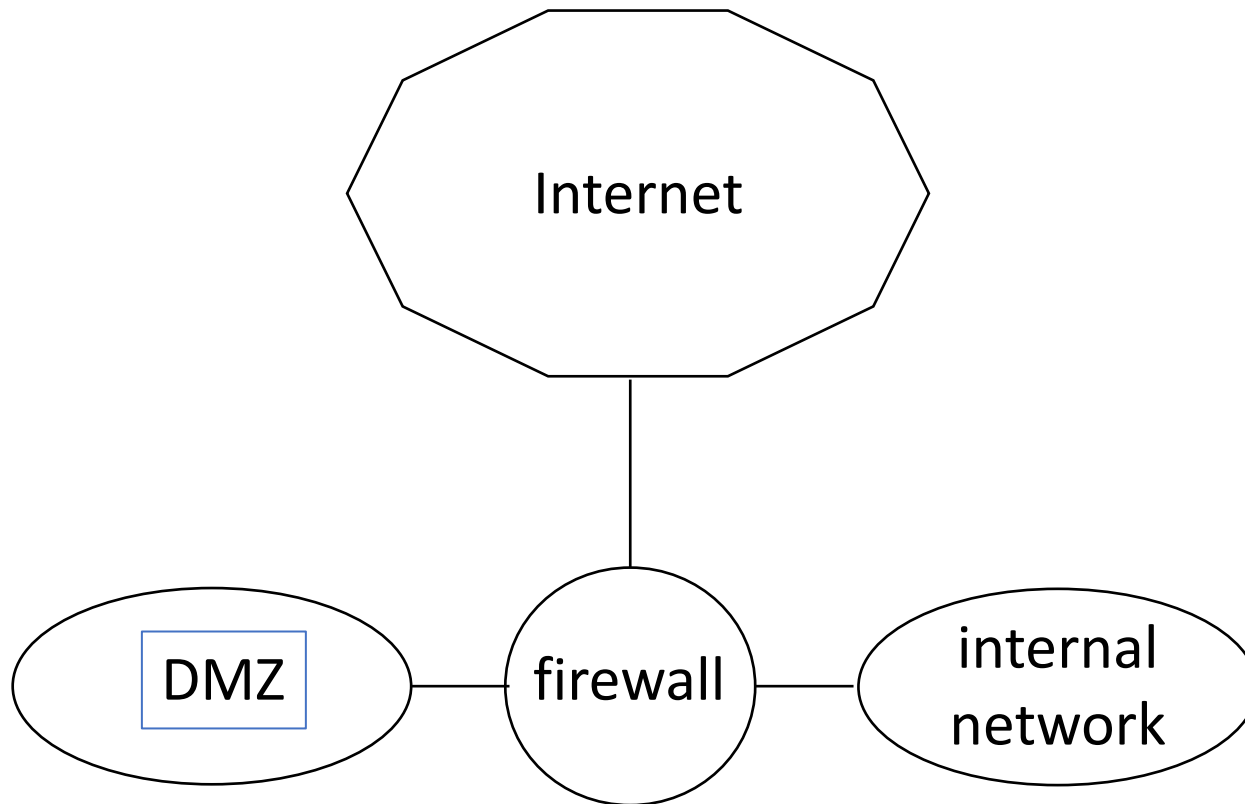
One dual-homed firewall that routes messages to internal network or DMZ as appropriate

Another Setup of DMZ

Internet

internal
network

outer
firewall

DMZ
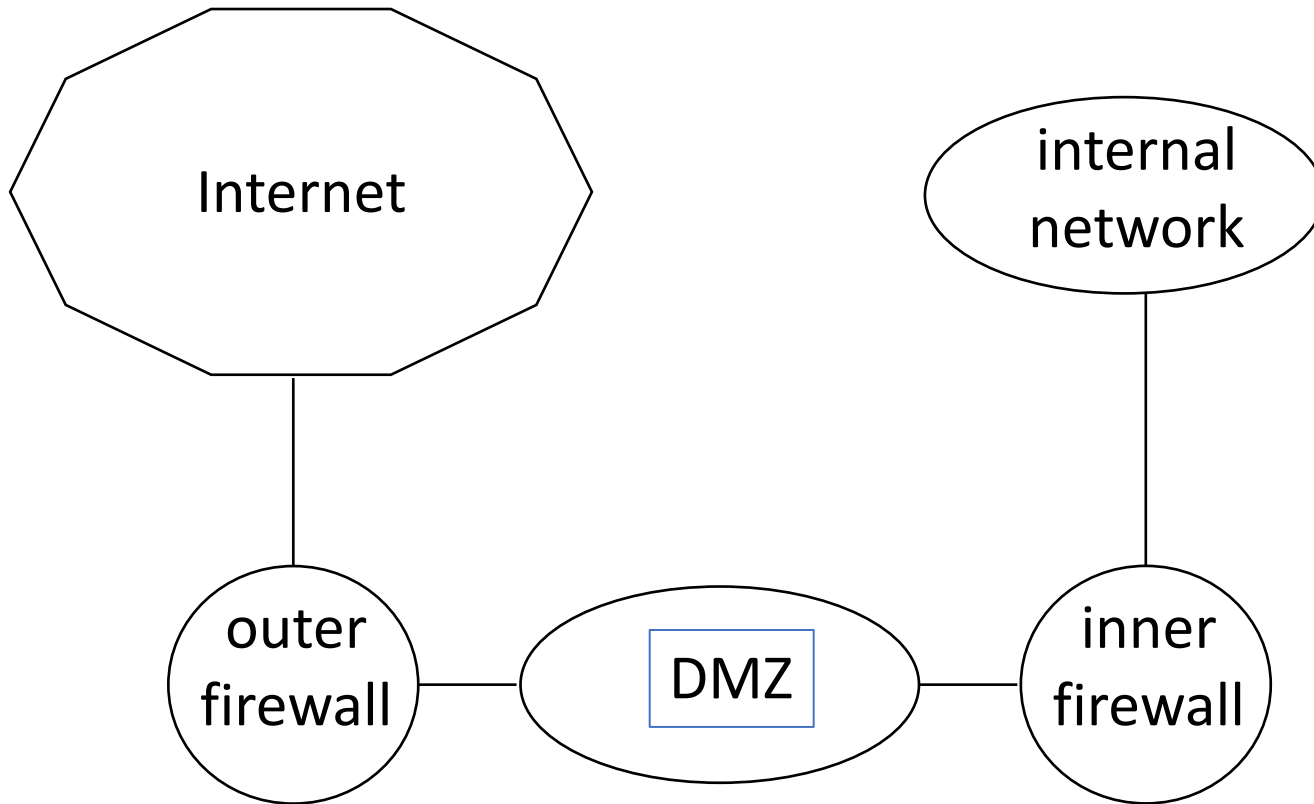
inner
firewall

Two firewalls, one (outer firewall) connected to the Internet, the other (inner firewall) connected to internal network, and the DMZ is between the firewalls