March 4, 2014

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

```plaintext
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 $sum \leq z$
  – This is false ($sum = \{ x \}$ dominates $z = \text{Low}$)
Infinite Loops

```plaintext
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 $y$

March 4, 2014

ECS 235B Winter Quarter 2014
Semaphores

Use these constructs:

```plaintext
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

```plaintext
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
  - $\text{shared}(S)$: set of shared variables read
    - Idea: information flows out of variables in $\text{shared}(S)$
  - $\text{fglb}(S)$: glb of assignment targets following $S$
  - So, requirement is $\text{shared}(S) \leq \text{fglb}(S)$
- $\text{begin } S_1; \ldots; S_n \text{ end}$
  - All $S_i$ must be secure
  - For all $i$, $\text{shared}(S_i) \leq \text{fglb}(S_i)$
begin
\[ x := y + z; \quad (* \ S_1 \ *) \]
\[ \text{wait}(\text{sem}); \quad (* \ S_2 \ *) \]
\[ a := b \times c - x; \quad (* \ S_3 \ *) \]
end

• Requirements:
  
  – \( \text{lub}(y, z) \leq x \)
  
  – \( \text{lub}(b, c, x) \leq a \)
  
  – \( \text{sem} \leq a \)

  • Because \( \text{fglb}(S_2) = a \) and \( \text{shared}(S_2) = \text{sem} \)
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 $S_1, \ldots, S_n$ 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

\[
\text{while } i < n \text{ do begin}
\]

\[
a[i] := \text{item}; \quad (* \ S_1 \ *) \\
\text{wait}(\text{sem}); \quad (* \ S_2 \ *) \\
i := i + 1; \quad (* \ S_3 \ *) \\
\text{end}
\]

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

cobegin

\[
\begin{align*}
  x & := y + z; \quad (* \ S_1 \ *) \\
  a & := b \times c - y; \quad (* \ S_2 \ *)
\end{align*}
\]

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, \( x \leq y \)
    \[
    \text{if } x = 1 \text{ then } y := a;
    \]
  – 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:
    
    \[
    \text{if } PC \leq x \text{ then } x := x + 1 \text{ else skip}
    \]

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

- if' \( x = 0 \) then goto \( n \) else \( x := x - 1 \)
  (branch without saving PC on stack)
  - Same as:
    \[
    \text{if } x = 0 \text{ then}
    \]
    \[
    \text{if } x \leq PC \text{ then } PC := n \text{ else skip}
    \]
    \[
    \text{else}
    \]
    \[
    \text{if } PC \leq x \text{ then } x := x - 1 \text{ else skip}
    \]
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. \textit{if } x = 0 \textit{ then goto 4 } \textit{ else } x := x - 1 \\
2. \textit{if } z = 0 \textit{ then goto 6 } \textit{ else } z := z - 1 \\
3. \textit{halt} \\
4. z := z + 1 \\
5. \textit{return} \\
6. y := y + 1 \\
7. \textit{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>Low $\leq x$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Low</td>
<td>—</td>
<td></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>$PC \leq y$</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 $z$ to Low
  - if $x = 0$ then $z := 1$ sets $z$ to 1 and $z$ to $x$
  - So on exit, $y = 0$

• $x = 1$
  - $z := 0$ sets $z$ to Low
  - if $z = 0$ then $y := 1$ sets $y$ to 1 and checks that $\text{lub}\{\text{Low}, z\} \leq y$
  - 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 \text{if } x = 0 \text{ then } z := 1, z raised to x whether or not \( x = 0 \)
  - Certification check in next statement, that \( z \leq y \), fails, as \( z = x \) from previous statement, and \( y \leq x \)
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 $z = z_0 \cdots z_{k-1}$

```
x := 1; atmp := a;
for i := 0 to k-1 do begin
  if $z_i$ = 1 then
    x := (x * atmp) mod n;
  atmp := (atmp * atmp) mod n;
end
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
Virtual Machine

• Program that simulates hardware of a machine
  – Machine may be an existing, physical one or an abstract one

• Why?
  – Existing OSes do not need to be modified
    • Run under VMM, which enforces security policy
    • Effectively, VMM is a security kernel
VMM as Security Kernel

• VMM deals with subjects (the VMs)
  – Knows nothing about the processes within the VM
• VMM applies security checks to subjects
  – By transitivity, these controls apply to processes on VMs
• Thus, satisfies rule of transitive confinement
Example 1: KVM/370

- KVM/370 is security-enhanced version of VM/370 VMM
  - Goal: prevent communications between VMs of different security classes
  - Like VM/370, provides VMs with minidisks, sharing some portions of those disks
  - Unlike VM/370, mediates access to shared areas to limit communication in accordance with security policy
Example 2: VAX/VMM

- Can run either VMS or Ultrix
- 4 privilege levels for VM system
  - VM user, VM supervisor, VM executive, VM kernel (both physical executive)
- VMM runs in physical kernel mode
  - Only it can access certain resources
- VMM subjects: users and VMs
Example 2

• VMM has flat file system for itself
  – Rest of disk partitioned among VMs
  – VMs can use any file system structure
    • Each VM has its own set of file systems
  – Subjects, objects have security, integrity classes
    • Called access classes
  – VMM has sophisticated auditing mechanism
Problem

• Physical resources shared
  – System CPU, disks, etc.

• May share logical resources
  – Depends on how system is implemented

• Allows covert channels
Sandboxes

- An environment in which actions are restricted in accordance with security policy
  - Limit execution environment as needed
    - Program not modified
    - Libraries, kernel modified to restrict actions
  - Modify program to check, restrict actions
    - Like dynamic debuggers, profilers
Examples Limiting Environment

• Java virtual machine
  – Security manager limits access of downloaded programs as policy dictates

• Sidewinder firewall
  – Type enforcement limits access
  – Policy fixed in kernel by vendor

• Domain Type Enforcement
  – Enforcement mechanism for DTEL
  – Kernel enforces sandbox defined by system administrator
Modifying Programs

• Add breakpoints or special instructions to source, binary code
  – On trap or execution of special instructions, analyze state of process

• Variant: *software fault isolation*
  – Add instructions checking memory accesses, other security issues
  – Any attempt to violate policy causes trap
Example: Janus

- Implements sandbox in which system calls checked
  - *Framework* does runtime checking
  - *Modules* determine which accesses allowed

- Configuration file
  - Instructs loading of modules
  - Also lists constraints
Configuration File

# basic module
basic

# define subprocess environment variables
putenv IFS=\"\t\n\" PATH=/sbin:/bin:/usr/bin TZ=PST8PDT

# deny access to everything except files under /usr
path deny read,write *
path allow read,write /usr/*
# allow subprocess to read files in library directories
# needed for dynamic loading
path allow read /lib/* /usr/lib/* /usr/local/lib/*
# needed so child can execute programs
path allow read,exec /sbin/* /bin/* /usr/bin/*
How It Works

- Framework builds list of relevant system calls
  - Then marks each with allowed, disallowed actions
- When monitored system call executed
  - Framework checks arguments, validates that call is allowed for those arguments
    - If not, returns failure
    - Otherwise, give control back to child, so normal system call proceeds
Use

- Reading MIME Mail: fear is user sets mail reader to display attachment using Postscript engine
  - Has mechanism to execute system-level commands
  - Embed a file deletion command in attachment …
- Janus configured to disallow execution of any subcommands by Postscript engine
  - Above attempt fails
Sandboxes, VMs, and TCB

• Sandboxes, VMs part of trusted computing bases
  – Failure: less protection than security officers, users believe
  – “False sense of security”

• Must ensure confinement mechanism correctly implements desired security policy
Covert Channels

- Shared resources as communication paths
- *Covert storage channel* uses attribute of shared resource
  - Disk space, message size, etc.
- *Covert timing channel* uses temporal or ordering relationship among accesses to shared resource
  - Regulating CPU usage, order of reads on disk
Example Storage Channel

- 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
Example Timing Channel

- System has two VMs
  - Sending machine $S$, receiving machine $R$
- To send:
  - For 0, $S$ immediately relinquishes CPU
    - For example, run a process that instantly blocks
  - For 1, $S$ uses full quantum
    - For example, run a CPU-intensive process
- $R$ measures how quickly it gets CPU
  - Uses real-time clock to measure intervals between access to shared resource (CPU)
Example Covert Channel

- Uses ordering of events; does not use clock
- Two VMs sharing disk cylinders 100 to 200
  - SCAN algorithm schedules disk accesses
  - One VM is High \((H)\), other is Low \((L)\)
- Idea: \(L\) will issue requests for blocks on cylinders 139 and 161 to be read
  - If read as 139, then 161, it’s a 1 bit
  - If read as 161, then 139, it’s a 0 bit
How It Works

- $L$ issues read for data on cylinder 150
  - Relinquishes CPU when done; arm now at 150
- $H$ runs, issues read for data on cylinder 140
  - Relinquishes CPU when done; arm now at 140
- $L$ runs, issues read for data on cylinders 139 and 161
  - Due to SCAN, reads 139 first, then 161
  - This corresponds to a 1
- To send a 0, $H$ would have issued read for data on cylinder 160
Analysis

• Timing or storage?
  – Usual definition ⇒ storage (no timer, clock)

• Modify example to include timer
  – $L$ uses this to determine how long requests take to complete
  – Time to seek to 139 < time to seek to 161 ⇒ 1; otherwise, 0

• Channel works same way
  – Suggests it’s a timing channel; hence our definition