ECS 235B Module 53
Detecting Covert Channels
Detection

• Manner in which resource is shared controls who can send, receive using that resource
  • Noninterference
  • Shared Resource Matrix Methodology
  • Information flow analysis
  • Covert flow trees
Noninterference

• View “read”, “write” as instances of information transfer
• Then two processes can communicate if information can be transferred between them, even in the absence of a direct communication path
  • A covert channel
  • Also sounds like interference ...
Example: SAT

- Secure Ada Target, multilevel security policy
- Approach:
  - $\pi(i, l)$ removes all instructions issued by subjects dominated by level $l$ from instruction stream $i$
  - $A(i, \sigma)$ state resulting from execution of $i$ on state $\sigma$
  - $\sigma.v(s)$ describes subject $s$’s view of state $\sigma$
- System is noninterference-secure iff for all instruction sequences $i$, subjects $s$ with security level $l(s)$, states $\sigma$,
  \[ A(\pi(i, l(s)), \sigma).v(s) = A(i, \sigma).v(s) \]
Theorem

• Version of the Unwinding Theorem

• Let $\Sigma$ be set of system states. A specification is noninterference-secure if, for each subject $s$ at security level $l(s)$, there exists an equivalence relation $\equiv: \Sigma \times \Sigma$ such that
  
  • for $\sigma_1, \sigma_2 \in \Sigma$, when $\sigma_1 \equiv \sigma_2$, $\sigma_1.v(s) = \sigma_2.v(s)$
  • for $\sigma_1, \sigma_2 \in \Sigma$ and any instruction $i$, when $\sigma_1 \equiv \sigma_2$, $A(i, \sigma_1) \equiv A(i, \sigma_2)$
  • for $\sigma \in \Sigma$ and instruction stream $i$, if $\pi(i, l(s))$ is empty, $A(\pi(i, l(s)), \sigma).v(s) = \sigma.v(s)$
Intuition

- System is noninterference-secure if:
  - Equivalent states have the same view for each subject
  - View remains unchanged if any instruction is executed
  - Instructions from higher-level subjects do not affect the state from the viewpoint of the lower-level subjects
Analysis of SAT

• Focus on object creation instruction and readable object set
• In these specifications:
  • s subject with security level \( l(s) \)
  • o object with security level \( l(o) \), type \( \tau(o) \)
  • \( \sigma \) current state
  • Set of existing objects listed in a global object table \( T(\sigma) \)
Specification 1

• object\_create:

\[ \sigma' = \text{object\_create}(s,o,l(o),t(o),\sigma) \land \sigma' \neq \sigma \]

\[ \iff \]

\[ o \notin T(\sigma) \land l(s) \leq l(o) \]

• The create succeeds if, and only if, the object does not yet exist and the clearance of the object will dominate the clearance of its creator
  • In accord with the “writes up okay” idea
Specification 2

- readable object set: set of existing objects that subject could read
  - $\text{can\_read}(s, o, \sigma)$ true if in state $\sigma$, $o$ is of a type that $s$ can read (ignoring permissions)
- $o \notin \text{readable}(s, \sigma) \iff \left[ o \notin T(\sigma) \lor \neg(l(o) \leq l(s)) \lor \neg(\text{can\_read}(s, o, \sigma)) \right]$
- Can’t read a nonexistent object, one with a security level that the subject’s security level does not dominate, or object of the wrong type
Specification 3

• SAT enforces tranquility
  • Adding object to readable set means creating new object
• Add to readable set:
  \[
  [o \notin \text{readable}(s, \sigma) \land o \in \text{readable}(s, \sigma')] \iff \\
  [\sigma' = \text{object}_\text{create}(s, o, l(o), \tau(o), \sigma) \land o \notin T(\sigma) \land l(s') \leq l(o) \leq l(s) \land \\
  \text{can}_\text{read}(s, o, \sigma')]
  \]
• Says object must be created, levels and discretionary access controls set properly
Check for Covert Channels

• $\sigma_1$, $\sigma_2$ the same except:
  • $o$ exists only in latter
  • $\neg(l(o) \leq l(s))$

• Specification 2:
  • $o \notin \text{readable}(s, \sigma_1)$ \{ $o$ doesn’t exist in $\sigma_1$ \}
  • $o \notin \text{readable}(s, \sigma_2)$ \{ $\neg(l(o) \leq l(s))$ \}

• Thus $\sigma_1 \equiv \sigma_2$
  • Condition 1 of theorem holds
Continue Analysis

• $s'$ issues command to create $o$:
  • with $l(o) = l(s)$; and
  • of type with $\text{can\_read}(s, o, \sigma_1')$
    • $\sigma_1'$ state after $\text{object\_create}(s', o, l(o), \tau(o), \sigma_1)$

• Specification 1
  • $\sigma_1'$ differs from $\sigma_1$ with $o$ in $T(\sigma_1)$

• New entry satisfies:
  • $\text{can\_read}(s, o, \sigma_1')$
  • $l(s') \leq l(o) \leq l(s)$, where $s'$ created $o$
Continue Analysis

• \( o \) exists in \( \sigma_2 \) so:

\[
\sigma_2' = \text{object\_create}(s', o, \sigma_2) = \sigma_2
\]

• But this means

\[
\neg \left[ A(\text{object\_create}(s', o, l(o), \tau(o), \sigma_2), \sigma_2) \equiv A(\text{object\_create}(s', o, l(o), \tau(o), \sigma_1), \sigma_1) \right]
\]

• Because create fails in \( \sigma_2 \) but succeeds in \( \sigma_1 \)

• So condition 2 of theorem fails

• This implies a covert channel as system is not noninterference-secure
Example Exploit

• To send 1:
  • High subject creates high object
  • Recipient tries to create same object but at low
    • Creation fails, but no indication given
  • Recipient gives different subject type permission to read, write object
    • Again fails, but no indication given
  • Subject writes 1 to object, reads it
    • Read returns nothing
Example Exploit

• To send 0:
  • High subject creates nothing
  • Recipient tries to create same object but at low
    • Creation succeeds as object does not exist
  • Recipient gives different subject type permission to read, write object
    • Again succeeds
  • Subject writes 1 to object, reads it
    • Read returns 1
Use

• Can analyze covert storage channels
  • Noninterference techniques reason in terms of security levels (attributes of objects)

• Covert timing channels much harder
  • You would have to make ordering an attribute of the objects in some way
SRMM

• Shared Resource Matrix Methodology
• Goal: identify shared channels, how they are shared
• Steps:
  • Identify all shared resources, their visible attributes [rows]
  • Determine operations that reference (read), modify (write) resource [columns]
  • Contents of matrix show how operation accesses the resource
Example

- Multilevel security model
- File attributes:
  - existence, owner, label, size
- File manipulation operations:
  - read, write, delete, create
    - create succeeds if file does not exist; gets creator as owner, creator’s label
    - others require file exists, appropriate labels
- Subjects:
  - High, Low
## Shared Resource Matrix

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
<th>delete</th>
<th>create</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>existence</strong></td>
<td>R</td>
<td>R</td>
<td>R, M</td>
<td>R, M</td>
</tr>
<tr>
<td><strong>owner</strong></td>
<td></td>
<td></td>
<td>R</td>
<td>M</td>
</tr>
<tr>
<td><strong>label</strong></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>M</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>R</td>
<td></td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>
Covert Storage Channel

• Properties that must hold for covert storage channel:
  1. Sending, receiving processes have access to same attribute of shared object;
  2. Sender can modify that attribute;
  3. Receiver can reference that attribute; and
  4. Mechanism for starting processes, properly sequencing their accesses to resource
Example

• Consider attributes with both R, M in rows
• Let High be sender, Low receiver
• *create* operation both references, modifies existence attribute
  • Low can use this due to semantics of *create*
• Need to arrange for proper sequencing accesses to existence attribute of file (shared resource)
Use of Channel

• 3 files: *ready*, *done*, *1bit*

• Low creates *ready* at High level

• High checks that file exists
  • If so, to send 1, it creates *1bit*; to send 0, skip
  • Delete *ready*, create *done* at High level

• Low tries to create *done* at High level
  • On failure, High is done
  • Low tries to create *1bit* at level High

• Low deletes *done*, creates *ready* at High level
Covert Timing Channel

• Properties that must hold for covert timing channel:
  1. Sending, receiving processes have access to same *attribute* of shared object;
  2. Sender, receiver have access to a time reference (wall clock, timer, event ordering, ...);
  3. Sender can control timing of detection of change to that attribute by receiver;
     and
  4. Mechanism for starting processes, properly sequencing their accesses to resource
Example

• Revisit variant of KVM/370 channel
  • Sender, receiver can access ordering of requests by disk arm scheduler (attribute)
  • Sender, receiver have access to the ordering of the requests (time reference)
  • High can control ordering of requests of Low process by issuing cylinder numbers to position arm appropriately (timing of detection of change)
  • So whether channel can be exploited depends on whether there is a mechanism to (1) start sender, receiver and (2) sequence requests as desired
Uses of SRM Methodology

• Applicable at many stages of software life cycle model
  • Flexibility is its strength

• Used to analyze Secure Ada Target
  • Participants manually constructed SRM from flow analysis of SAT model
  • Took transitive closure
  • Found 2 covert channels
    • One used assigned level attribute, another assigned type attribute
Summary

• Methodology comprehensive but incomplete
  • How to identify shared resources?
  • What operations access them and how?

• Incompleteness a benefit
  • Allows use at different stages of software engineering life cycle

• Incompleteness a problem
  • Makes use of methodology sensitive to particular stage of software development
Information Flow Analysis

• When exception occurs due to value of variable, information leaks about the value – a covert channel
  • Same for synchronization and IPC primitives, because one process controls when it sends message or blocks to receive one
  • Shared variables are *not* covert channel as they are intended to share values

• Method for identifying covert storage channels in source code
  • Assertion: these arise when processes can view, alter kernel variables
  • So identify these variables
    • May be directly referenced or indirectly referenced via system calls
Step 1

• Identify kernel functions, processes for analysis
  • Processes function at privileged level, but carry out actions for ordinary users
  • Ignore those executing on behalf of administrators (they can leak information directly)
  • Same with system calls available only to system administrator
Step 2

• Identify kernel variables user process can view and/or alter
  • Process must control how variable is altered
  • Process must be able to detect that variable was altered

• Detection criteria
  • Value of a variable is obtained from system call
  • Calling process can detect at least 2 different states of that variable

• Examples
  • If system call assigns fixed value to a particular variable, process cannot control how that variable is altered
  • If value of x causes an error, state of x can be determined from the error indicator
Directly vs. Indirectly Visible

x directly visible to caller as it is returned directly to caller

\[ x := \text{func}(abc, \text{def}); \]
\[ \text{if } x = 0 \text{ then} \]
\[ \quad x := x + 10; \]
\[ \text{return } x; \]

y not directly visible to caller, but indirectly visible as its state observed through z

\[ y := \text{func}(abc, \text{def}); \]
\[ \text{if } y = 0 \text{ then} \]
\[ \quad z := 1; \]
\[ \text{else} \]
\[ \quad z := 0; \]
\[ \text{return } z; \]
Step 3

• Analyze variables looking for covert channels
  • Use method similar to that of SRM
  • Discard primitives associated with variables that can *only* be altered or *only* be viewed
  • Assume recipient’s clearance does not dominate sender’s, and compare resulting primitives to model of access control
Covert Flow Trees

• Information flow through shared resources modeled using tree
  • Flow paths identified, and analyzed to see if each is legitimate

• 5 types of nodes
  • *Goal symbols*: states that must exist for information to flow
  • *Operation symbol*: symbol representing primitive operation
  • *Failure symbol*: information cannot be sent along the path containing it
  • *And symbol*: goal reached when these hold for all children
    • If the child is a goal, then the goal is reached; and
    • The child is an operation
  • *Or symbol*: goal reached when either of these hold for any children
    • If the child is a goal, then the goal is reached; or
    • The child is an operation
More on Goal Symbols

- *Modification goal*: reached when attribute is modified
- *Recognition goal*: reached when modification of attribute is detected
- *Direct recognition goal*: reached when subject can detect modification of attribute by direct reference or calling a function that returns it
- *Inferred recognition goal*: reached when subject can detect modification of attribute without directly referencing it or calling a function that references attribute directly
- *Inferred-via goal*: reached when information passed from one attribute to others using specified primitive operation
- *Recognized-new-state goal*: reached when an attribute that was modified when information passed using it is specified by inferred-via goal
Example Program

procedure Lockfile(f: file): boolean; (* lock file if not locked; return *)
begin (* false if locked, true otherwise *)
    if not f.locked and empty(f.inuse) then
        f.locked := true;
    Lockfile := not f.locked;
end;

procedure Unlockfile(f: file); (* unlock file *)
begin
    if f.locked then
        f.locked := false;
end;

function Filelocked(f: file): boolean; (* return state of file locking *)
begin
    Filelocked := f.locked;
end;
Example Program

procedure Openfile(f: file); (* open file if not locked and *)
begin (* permissions allow it *)
    if not f.locked and read_access(process_id, f) then
        (* add the process ID to the inuse set *)
        f.inuse := f.inuse + process_id;
end;

function Fileopened(f: file): boolean; (* if permissions allow process to read file, *)
begin (* say if open; else return random value. *)
    if not read_access(process_id, f) then
        Fileopened := random(true, false);
    else
        Fileopened := not isempty(f.inuse);
end;
Step 1

• Determine attributes that primitive operations reference, modify, return

<table>
<thead>
<tr>
<th></th>
<th>Lockfile</th>
<th>Unlockfile</th>
<th>Filelocked</th>
<th>Openfile</th>
<th>Fileopened</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>locked,inuse</td>
<td>locked</td>
<td>locked</td>
<td>locked,inuse</td>
<td>inuse</td>
</tr>
<tr>
<td>modify</td>
<td>locked</td>
<td>Ø</td>
<td>Ø</td>
<td>inuse</td>
<td>Ø</td>
</tr>
<tr>
<td>return</td>
<td>Ø</td>
<td>Ø</td>
<td>locked</td>
<td>Ø</td>
<td>inuse</td>
</tr>
</tbody>
</table>
Step 2

• Construct the flow tree; controlled by type of goal
• Construction ends when all paths terminate in either operation symbol of failure symbol
  • If loops occur, a parameter defines number of times path may be traversed
Step 2 (con’t)

• Topmost goal: requires attribute be modified and the modification be recognized
  • 1 child (and) with 2 goals (modification, recognition goal symbols)

• Modification goal: requires primitive operation to modify attribute
  • 1 child (or) with 1 child operation symbol per operation for all operations that modify attribute

• Recognition goal: subject directly recognize or infer change in attribute
  • 1 child (or) with 2 children (direct recognition, inferred recognition goals)
Step 2 (con’t)

• Direct recognition goal: operation accesses attribute
  • 1 child (or) with 1 child operation symbol per operation for each operation that returns attribute

• Inferred recognition goal: modification referred on basis of 1 or more attributes
  • 1 child (or) with 1 child inferred-via symbol per operation for each operation that references an attribute and modifies an attribute

• Inferred-via goal: value of attribute inferred via some operation and new state of attribute recognized
  • 1 child (and) with 2 children (operation, recognize-new-state goal symbols)

• Recognize-new-state goal: value of attribute inferred via some operation and new state of attribute recognized, requiring a recognition goal for attribute
  • 1 child (or) and for each attribute enabling inference of modification of attribute in question, 1 child (recognition goal symbol)
Example: Goal State and Modification Branch

- The next few slides build covert flow tree for attribute *locked*

```
Example: Goal State and Modification Branch

- The next few slides build covert flow tree for attribute *locked*

```

```
Module 53

ECS 235B, Foundations of Computer and Information Security

```

```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Example: Recognition Branch

- Recognition of attribute *locked*
  - Direct recognition of attribute *locked*
    - Lockfile
  - Indirect recognition of attribute *locked*
    - Indirect attribute *locked* via attribute *inuse*
Example: Indirect Branch

Indirect attribute *locked*
via attribute *inuse*

- Openfile
- Recognition of attribute *inuse*
Example: Recognize New Goal State Branch

Recognition of attribute *inuse*

Direct recognition of attribute *inuse*

Indirect recognition of attribute *locked*

*Fileopened*

*FALSE*
Example: Analysis

• Put those parts of the tree together in the obvious way
• First list: \( ((\text{Lockfile}, \ (\text{Unlockfile}))) \)
  • As both modify attribute \textit{locked} and lie on “modified” branch
• Second list: \( ((\text{Filelocked}), \ (\text{Openfile, Fileopened})) \)
  • From direct recognition of modification of \textit{inuse} attribute; second, from indirect recognition of modification of attribute \textit{locked}
• These result in 4 paths of communication:
  • \textit{Lockfile} followed by \textit{Filelocked}
  • \textit{Unlockfile} followed by \textit{Filelocked}
  • \textit{Lockfile} followed by \textit{Openfile}, then \textit{Fileopened}
  • \textit{Unlockfile} followed by \textit{Openfile}, then \textit{Fileopened}
Example: Analysis

• First two sequences in combination represent direct covert storage channel
  • *High* process transmits information to *Low* process by locking, unlocking file

• Last two sequences represent indirect covert storage channel
  • High process locks file to send 0, unlocks to send 1
  • Low process tries to open the file, then uses Fileopened to see if it succeeded
  • If opened, file was not locked and it’s a 1; if not opened, file is locked, and it’s a 0
Summary

• Covert flow trees, SRM come from idea that covert channels require shared resources that one process can modify and another view
• Both can be used at any point in life cycle
• Covert flow trees identify explicit sequences of operations causing information to flow
  • SRM identifies *channels*, not sequences of operations