ECS 289M Lecture 24

May 26, 2006

Computer Virus

- Program that inserts itself into one or more files and performs some action
  - *Insertion phase* is inserting itself into file
  - *Execution phase* is performing some (possibly null) action
- Insertion phase *must* be present
  - Need not always be executed
  - Lehigh virus inserted itself into boot file only if boot file not infected

Pseudocode

```
beginvirus:
  if spread-condition then begin
    for some set of target files do begin
      if target is not infected then begin
        determine where to place virus instructions
        copy instructions from beginvirus to endvirus
          into target
        alter target to execute added instructions
      end;
    end;
  end;
  perform some action(s)
  goto beginning of infected program
endvirus:
```


Trojan Horse Or Not?

- Yes
  - Overt action = infected program's actions
  - Covert action = virus' actions (infect, execute)
- No
  - Overt purpose = virus' actions (infect, execute)
  - Covert purpose = none
- Semantic, philosophical differences
  - Defenses against Trojan horse also inhibit computer viruses

Computer Worms

- A program that copies itself from one computer to another
- Origins: distributed computations
  - Schoch and Hupp: animations, broadcast messages
  - Segment: part of program copied onto workstation
  - Segment processes data, communicates with worm's controller
  - Any activity on workstation caused segment to shut down

Example: Internet Worm of 1988

- Targeted Berkeley, Sun UNIX systems
  - Used virus-like attack to inject instructions into running program and run them
  - To recover, had to disconnect system from Internet and reboot
  - To prevent re-infection, several critical programs had to be patched, recompiled, and reinstalled
- Analysts had to disassemble it to uncover function
- Disabled several thousand systems in 6 or so hours

Example: Christmas Worm

- Distributed in 1987, designed for IBM networks
- Electronic letter instructing recipient to save it and run it as a program
  - Drew Christmas tree, printed “Merry Christmas!”
  - Also checked address book, list of previously received email and sent copies to each address
- Shut down several IBM networks
- Really, a macro worm
  - Written in a command language that was interpreted

Theory of Detection

- Can we write a program to detect all computer viruses precisely, without error?
- YES!!!
  - What follows is from Dr. Alan Soloman (Dr. Solly to most folks)

The Perfect Antivirus

I shall now give you, free of charge, an antivirus that if used correctly, detects all past, present and future viruses, never gives a false alarm, and has a zero cost. Skeptical? Then watch carefully ...

```
type P1.BAT
Echo %1 is infected by a virus!!!
```

You'll agree, I think, that P1.BAT will detect all past present and future viruses. That alone meets the “mathematically impossible” task!

But, I hear you thinking, aren’t there rather a lot of false alarms? Well, you didn't say you wanted a low false alarm rate....

Not Good Enough

OK, OK. I'm used to projects where the user specification changes in the middle. Never mind. I can deal with the false alarms ...

```
P2.BAT
Echo %1 is NOT infected by a virus!!!
```

You’ll agree, I think, that P2.BAT will never, ever, tell you that you have a virus when you don’t. Of course, it has a pretty poor detection rate. I admit that.

So Here It Is!

But I can fix it. See here ...

```
PERFECT.BAT
Echo Is %1 a virus? (Y/N)
```

If the user types ‘Y’, you run P1. If the user types ‘N’, you run P2.

Remember what I promised you? An antivirus that *if used correctly*, detects all past, present and future viruses, never gives a false alarm, and has a zero cost.

Moral of All This?

All very amusing, but what can we learn from this?

1. If something is superb at detecting viruses, it's no use if it gives a lot of false alarms.
2. Anything that relies on the user to make a correct decision, on matters that he is not likely to be able to decide about, is useless.
3. You can receive something that is *exactly* what the salesman promised to deliver, and it's nevertheless useless.

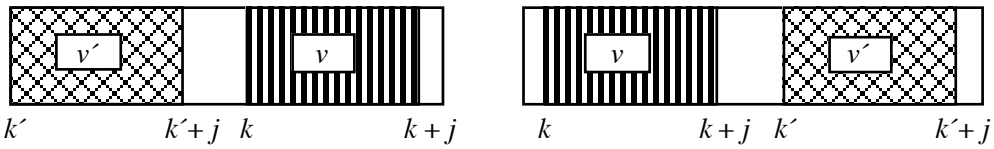
OK, Back to Math …

- Is there a single algorithm that detects computer viruses precisely?
  - Need to define viruses in terms of Turing machines
  - See if we can map the halting problem into that algorithm

Step 1: Virus

- T Turing machine
  - s_v distinguished state of T
- V sequence of symbols on machine tape
- For every $v \in V$, when T lies at the beginning of v in tape square k, suppose that after some number of instructions are executed, a sequence $v' \in V$ lies on the tape beginning at location k', where either $k+|v| \leq k'$ or $k'+|v| \leq k$.
- (T, V) is a *viral set* and the elements of V are computer viruses.

In A Picture



- Virus *v* can copy another element of *V* either before or after itself on the tape
  - May not overwrite itself
  - Before at left, after at right

Overview of Argument

- Arbitrary *T*, sequence *S* of symbols on tape
- Construct second Turing machine *T′*, tape *V*, such that when *T* halts on *S*, *V* and *T′* create copy of *S* on tape
- *T′* replicates *S* iff *T* halts on *S*
  - Recall replicating program is a computer virus
- So there is a procedure deciding if (*T′*, *V*) is a viral set iff there is a procedure that determines if *T* halts on *S*
  - That is, if the halting problem is solvable

Theorem

- It is undecidable whether an arbitrary program contains a computer virus
- Proof:
  - T defines Turing machine
  - V defines sequence of tape symbols
  - $A, B \in M$ (tape symbols)
  - $q_i \in K$ for $i \geq 1$ (states)
  - a, b, i, j non-negative integers
  - δ: $K \times M \rightarrow K \times M \times \{L, R, -\}$ (transition function; – is no motion)

Proof

- Abbreviation for δ:

  $\delta(q_a, y) = (q_a, y, L)$ when $y \neq A$

  means all definitions of d where:
  - first element (current state) is q_a
  - second element (tape symbol) is anything other than A
  - third element is L (left head motion)

Abbreviations

- $LS(q_a, x, q_b)$
  - In state q_a, move head left until square with symbol x
  - Enter state q_b
  - Head remains over symbol x
- $RS(q_a, x, q_b)$
  - In state q_a, move head right until square with symbol x
  - Enter state q_b
  - Head remains over symbol x

Abbreviations

- $LS(q_a, x, q_b)$

  $\delta(q_a, x) = (q_b, x, -)$

  $\delta(q_a, y) = (q_a, y, L)$ when $y \neq x$
- $RS(q_a, x, q_b)$

  $\delta(q_a, x) = (q_b, x, -)$

  $\delta(q_a, y) = (q_a, y, R)$ when $y \neq x$

Abbreviation

- *COPY*(q_a, x, y, z, q_b)
  - In state q_a, move head right until square with symbol x
  - Copy symbols on tape until next square with symbol y
  - Place copy after first symbol z following y
  - Enter state q_b

Idea of Actions

- Put marker (A) over initial symbol
- Move to where to write it (*B*)
- Write it and mark location of next symbol (move *B* down one)
- Go back and overwrite marker *A* with symbol
- Iterate until *V* copied
  - Note: *A*, *B* symbols that do not occur in V

Abbreviation

$RS(q_a, x, q_{a+i})$

$\delta(q_{a+i}, x) = (q_{a+i+1}, A, -)$

- Move head over x, replace with marker A

$RS(q_{a+i+1}, y, q_{a+i+2})$

$RS(q_{a+i+2}, z, q_{a+i+3})$

- Skip to where segment is to be copied

$\delta(q_{a+i+3}, z) = (q_{a+i+4}, z, R)$

$\delta(q_{a+i+4}, u) = (q_{a+i+5}, B, -)$ for any $u \in M$

- Mark next square with B

More

- $LS(q_{a+i+5}, A, q_{a+i+6})$
- $\delta(q_{a+i+6}, A) = (q_{a+i+7}, x, -)$
  - Put x (clobbered by A) back
- $\delta(q_{a+i+7}, s_j) = (q_{a+i+5j+10}, A, R)$ for $s_j \neq y$
- $\delta(q_{a+i+7}, y) = (q_{a+i+8}, y, R)$
  - Overwrite symbol being copied (if last, enter new state)
- $RS(q_{a+i+5j+10}, B, q_{a+i+5j+11})$
- $\delta(q_{a+i+5j+11}, B) = (q_{a+i+5j+12}, s_j, R)$
  - Make copy of symbol

More

$\delta(q_{a+i+5j+12}, u) = (q_{a+i+5j+13}, B, -)$
- Mark where next symbol goes

$LS(q_{a+i+5j+13}, A, q_{a+i+5j+14})$

$\delta(q_{a+i+5j+14}, A) = (q_{a+i+7}, s_j, R)$
- Copy back symbol

$RS(q_{a+i+8}, B, q_{a+i+9})$

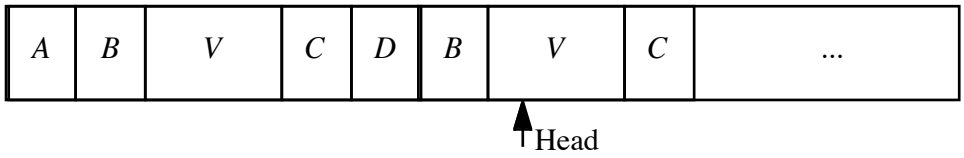
$\delta(q_{a+i+9}, B) = (q_b, y, -)$
- Write terminal symbol

Construction of T', V'

- Symbols of T': $M' = M \cup \{ A, B, C, D \}$
- States of T' :
  $K' = K \cup \{ q_a, q_b, q_c, q_d, q_e, q_f, q_g, q_h, q_H \}$
- q_a initial state of T'
- q_H halting state of T'
- $SIMULATE(q_f, T, q_h)$
  - Simulate execution of T on tape with head at current position, q_f, q_h in K' correspond to initial, terminal state of T

T'

- Let $V' = (A, B, V, C, D)$.
- Idea: copy V after D, run T on V, and if it finishes, copy V over results
- Then if T halts, (T', V) a viral set by definition



Running T in T'

$\delta(q_a, A) = (q_b, A, -)$

$\delta(q_a, y) = (q_H, y, -)$ for $y \neq A$

- Beginning, halting transitions

$COPY(q_b, B, C, D, q_c)$

- Copy V after D

$LS(q_c, A, q_d)$

$RS(q_d, D, q_e)$

$\delta(q_e, D) = (q_e, D, R)$

- Position head so T executes copy of V

Running T in T'

$\delta(q_e, B) = (q_f, B, R)$
- Position head after B at beginning of copy of V

SIMULATE(q_f, T, q_h)
- T runs on copy of V (execution phase)

LS(q_h, A, q_g)
- T finishes; go to beginning of T' tape

COPY(q_g, A, D, D, q_H)
- Copy initial contents of V over results of running T on V (reproduction phase)

Analysis

- If T halts on V, definition of “viral set” and “virus” satisfied
- If T never halts on V, V never recopied, and definition never satisfied
- Establishes result

More General Result

- **Theorem:** It is undecidable whether an arbitrary program contains malicious logic

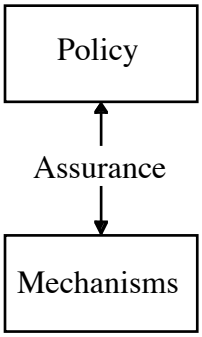
Basics of Assurance

- Trust
- Problems from lack of assurance
- Types of assurance
- Life cycle and assurance
- Waterfall life cycle model
- Other life cycle models

Trust

- *Trustworthy* entity has sufficient credible evidence leading one to believe that the system will meet a set of requirements
- *Trust* is a measure of trustworthiness relying on the evidence
- *Assurance* is confidence that an entity meets its security requirements based on evidence provided by applying assurance techniques

Relationships



Statement of requirements that explicitly defines the security expectations of the mechanism(s)

Provides justification that the mechanism meets policy through assurance evidence and approvals based on evidence

Executable entities that are designed and implemented to meet the requirements of the policy

Problem Sources

1. Requirements definitions, omissions, and mistakes
2. System design flaws
3. Hardware implementation flaws, such as wiring and chip flaws
4. Software implementation errors, program bugs, and compiler bugs
5. System use and operation errors and inadvertent mistakes
6. Willful system misuse
7. Hardware, communication, or other equipment malfunction
8. Environmental problems, natural causes, and acts of God
9. Evolution, maintenance, faulty upgrades, and decommissions

Examples

- Challenger explosion
  - Sensors removed from booster rockets to meet accelerated launch schedule
- Deaths from faulty radiation therapy system
  - Hardware safety interlock removed
  - Flaws in software design
- Bell V22 Osprey crashes
  - Failure to correct for malfunctioning components; two faulty ones could outvote a third
- Intel 486 chip
  - Bug in trigonometric functions

Role of Requirements

- *Requirements* are statements of goals that must be met
  - Vary from high-level, generic issues to low-level, concrete issues
- *Security objectives* are high-level security issues
- *Security requirements* are specific, concrete issues

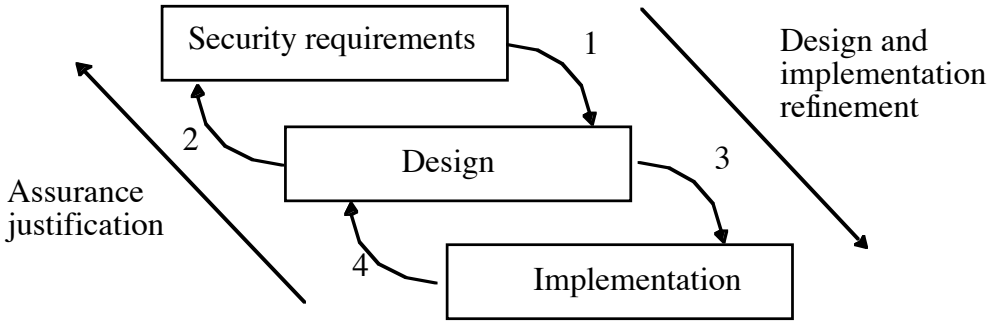
Types of Assurance

- *Policy assurance* is evidence establishing security requirements in policy is complete, consistent, technically sound
- *Design assurance* is evidence establishing design sufficient to meet requirements of security policy
- *Implementation assurance* is evidence establishing implementation consistent with security requirements of security policy

Types of Assurance

- *Operational assurance* is evidence establishing system sustains the security policy requirements during installation, configuration, and day-to-day operation
  - Also called *administrative assurance*

Life Cycle



Life Cycle

- Conception
- Manufacture
- Deployment
- Fielded Product Life

Conception

- Idea
  - Decisions to pursue it
- Proof of concept
  - See if idea has merit
- High-level requirements analysis
  - What does “secure” mean for this concept?
  - Is it possible for this concept to meet this meaning of security?
  - Is the organization willing to support the additional resources required to make this concept meet this meaning of security?

Manufacture

- Develop detailed plans for each group involved
  - May depend on use; internal product requires no sales
- Implement the plans to create entity
  - Includes decisions whether to proceed, for example due to market need