Security and Course Wrapup

Last Time

- Security theory
  - Access control matrix
  - Passwords
  - Encryption
- Security practice
  - Example successful attacks

Main Points

- Security practice
  - More example attacks
  - How to write an undetectable self-replicating virus
- Course wrapup

UNIX talk

- UNIX talk was an early version of Internet chat
  - For users logged onto same machine
- App was setuid root
  - Needed to write to everyone's terminal
- But it had a bug…
  - Signal handler for ctl-C

Netscape

- How do you pick a session key?
  - Early Netscape browser used time of day as seed to the random number generator
  - Made it easy to predict/break
- How do you download a patch?
  - Netscape offered patch to the random seed problem for download over Web, and from mirror sites
  - four byte change to executable to make it use attacker's key

Code Red/Nimda/Slammer

- Dictionary attack of known vulnerabilities
  - known Microsoft web server bugs, email attachments, browser helper applications, …
  - used infected machines to infect new machines
- Code Red:
  - designed to cause machines surf to whitehouse.gov simultaneously
- Nimda:
  - Left open backdoor on infected machines for any use
  - Infected ~ 400K machines
- Slammer:
  - Single UDP packet on MySQL port
  - Infected 100K+ vulnerable machines in under 10 minutes
- Million node botnets now common

More Examples

- Housekeys
- ATM keypad
- Automobile backplane
- Pacemakers

Thompson Virus

- Ken Thompson self-replicating program
  - installed itself silently on every UNIX machine, including new machines with new instruction sets

Add backdoor to login.c

- Step 1: modify login.c

  A:

  ```
  if (name == "ken") {
     don't check password;
     login ken as root;
  }
  ```
- Modification is too obvious; how do we hide it?

Hiding the change to login.c

- Step 2: Modify the C compiler

  B:

  ```
  if see trigger {
     insert A into the input stream
  }
  ```
- Add trigger to login.c

  /* gobblygook */
- Now we don't need to include the code for the backdoor in login.c, just the trigger
  - But still too obvious; how do we hide the modification to the C compiler?

Hiding the change to the compiler

- Step 3: Modify the compiler

  C:

  ```
  if see trigger2 {
     insert B and C into the input stream
  }
  ```
- Compile the compiler with C present
  - now in object code for compiler
- Replace C in the compiler source with trigger2

Compiler compiles the compiler

- Every new version of compiler has code for B,C included
  - as long as trigger2 is not removed
  - and compiled with an infected compiler
  - if compiler is for a completely new machine: cross-compiled first on old machine using old compiler
- Every new version of login.c has code for A included
  - as long as trigger is not removed
  - and compiled with an infected compiler

Question

- Can you write a self-replicating C program?
  - program that when run, outputs itself
    - without reading any input files!

```
char *buf =
        "char *buf = %c%s%c; main(){printf(buf, 34, buf, 34);}";
main() { printf(buf, 34, buf, 34); }
```

Security Lessons

- Hard to re-secure a machine after penetration
  - how do you know you've removed all the backdoors?
- Hard to detect if machine has been penetrated
  - Western Digital example
- Any system with bugs is vulnerable
  - and all systems have bugs: fingerd, ping of death, Code Red, nimda, …

Course Wrapup

Major Topics

- Protection
  - Kernel/user mode, system calls
- Concurrency
  - Threads, monitors, deadlock, scheduling
- Memory management
  - Address translation, demand paging
- File systems
  - Disk, flash, file layout, transactions

OS as Referee

- Protection
  - OS isolates apps from bugs or attacks in other apps
  - Pipes and files for interprocess communication
- CPU scheduling
  - OS decides which application thread is next onto the processor
- Memory allocation
  - OS decides how many memory frames given to each app
- File system
  - OS enforces security policy in accessing file data

OS as Illusionist

Physical Reality	*Abstraction*
Limited # of CPUs	Can assume near infinite # of processes/threads
CPU interrupts and time slicing	Each thread appears to run sequentially (at variable speed)
Limited physical memory	Near-infinite virtual memory
Apps share physical machine	Execution on virtual machine with isolation between apps
Computers can crash	Changes to file system are atomic and durable

OS as Glue

- Locks and condition variables
  - Not test&set instructions
- Named files and directories
  - Not raw disk block storage
- Pipes: stream interprocess communication
  - Not fixed size read/write calls
- Memory-mapped files
  - Not raw disk reads/writes

OS Trends and Future Directions

- Optimize for the computer's time
  => optimize for the user's time
- One processor => many
- One computer => server clusters
- Disk => solid state memory
- Operating systems at user level
  - Browsers, databases, servers, parallel runtimes

Advertisements

- CSE 452: Distributed Systems
  - How can we build scalable systems that work even though parts of the system can fail at any time?
- CSE 484: Security
  - How can we build systems that can withstand attack?
- CSE 444: Databases
  - How do we build systems that can manage giant amounts of data reliably and efficiently?
- CSE 461: Networks
  - How do we build protocols to allow reliable and efficient communication between computers?