Security & Trust

1. Terms
   a. Discretionary & mandatory access control
   b. trusting trust – what can be trusted?

2. Threat models
   a. What can an attacker do?
   b. Examples:
      i. inspect raw memory contents, modify raw memory contents externally
      ii. Sniff data off wires
      iii. Provide boot software
      iv. Send packets to network ports
      v. Run unprivileged programs as a user
   c. Why?
      i. Defines what range of behavior is being prevented
      ii. May exclude certain classes of attacks:
         1. timing attacks measure how long operations take to infer data, e.g. RSA keys
         2. Side channel attacks look at cache behavior (what lines are dirty) to infer operations
         3. Inducing processor errors: apply radiation or heat to trigger processor failure to bypass rules
            a. e.g. encrypted packets – can tell from length of packets & frequency that traffic is Facebook or Google even if destination and contents are encrypted
   d. Inktag: OS is untrusted
      i. What mean?
      ii. Can do arbitrary thigns:
         1. return from exit
         2. lie about return address from mmap()
         3. not write data in write(), return wrong data from read()
         4. launch wrong process on exec()
         5. Not deliver signals, or send wrong signal with signal()
      iii. Cannot do algorithmically hard things
         1. Guess secret keys
   3. Trusted computing base = the code (or hardware) you have to trust to execute correctly for security to be maintained
      a. Linux:
         i. Kernel
         ii. Loadable modules
iii. setuid root programs
iv. Config data for root programs
   1. scripts that launch programs as root
   2. Input files for root programs (could be corrupted)
v. Processor, memory, disk hardware, etc
b. Larger TCB == more code that must be bug-free to have security

4. Trusting trust:
   a. What code can you trust to execute correctly?
   b. Your operating system?
      i. How do you know it doesn’t have bugs?
      ii. Look at the code
         1. you don’t run the code – you compile it
         2. So you trust the compiler. Why?
            a. Look at the compiler code. But you don’t run the
               compiler code, you run the compiler
            b. Someone compiled the compiler you used to compile
               your compiler
               i. etc. ad infinitum
   c. Real example:
      i. Unix login program – accept fixed username/pwd, allow login
      ii. Code is not in program – in compiler
         1. if (program == login && line = “if strcmp(passwd,
            storedPwd)”
         2. replace with if (program == login && (line = “if
            strcmp(passwd, storedPwd) ||
            strcmp(passwd, “rootme”)”
      iii. But – could find by inspecting compiler.
         1. Instead, modify compiler:
            2. if “program = compiler” & line = “parse string”)
               a. instrt if (program == login && line = “if
                  strcmp(passwd, storedPwd)” ...
            3. Run once to produce a binary compiler
            4. Distribute this compiler
               a. Every time it compiles source, it will insert the back
                  door
               b. Source code distributed does not have
   d. Generally: what makes you trust code?
      i. You wrote the code
      ii. You read the code
         1. and it was pretty small
         2. and you understood it
      iii. You trust the publisher
         1. they have a lot to lose
         2. they have a good reputation
3. they gave you a money-backed guarantee
   iv. General principle:
      1. Smaller code is more trustworthy than bigger code

5. QUESTIONS
   a. What code do you trust?
   b. Why
   c. For Inktag:
      i. What are the ideas?
      ii. What are the abstractions?
      iii. What is the problem being solved
      iv. What is an Iago attack?

6. Inktag
   a. Key ideas:
      i. Trust
         1. Key idea 1: not trust OS
            a. It is not your code, you don’t really know what it does
            b. OS may have bugs, may have been root-kitted
         2. Key idea 2: trust application
            a. It is your code, you trust it
         3. Key idea 3: trust a hypervisor
            a. It is small, doesn’t do much, easier to make sure does
      ii. How cut out OS?
         1. Prevent from affecting control flow
            a. When does the OS affect control flow?
               i. Interrupts/traps: saves/restores registers
            b. How?
               i. Make hypervisor save /restore registers instead,
                  give OS fake registers (or real ones if you don’t
                  need privacy)
         2. Prevent OS from modifying data (integrity)
            a. How does OS access data?
               i. When running in privileged mode, read a
                  virtual address mapped to a physical page
                  containing application data
            b. How protect?
               i. Turn on write-protection for any virtual page
                  accessing app data when in ring 0 (kernel
                  mode)
         3. Prevent OS from reading data (secrecy)
            a. How does OS access data – same as integrity
            b. How prevent?
               i. Cannot make page invalid – OS has to copy
                  data around, swap to disk
ii. Encrypt instead: OS doesn’t need contents of page, just random bits.

iii. So: encrypt page before OS reads it (can do on entry to OS or by marking page invalid & doing it on access)

iii. How keep calling the OS?
   1. Provide unencrypted read for transition – the trampoline or shim
   2. Has data that doesn’t get encrypted, registers that can be saved/restored

7. InkTag abstractions
   a. Objects
      i. Definition
         1. range of memory mapped for a use; equivalent to Linux “Virtual Memory Areasd”
         2. Every offset iaddress nto range corresponds to same offset into the object
         3. Can be anonymous memory (just a range of virtual memory) or a memory mapped file
      ii. Naming
         1. Object Identifiers = OIDs
   b. s-page
      i. page that is part of an object
      ii. Protection:
         1. hashed & encrypted before made accessable to OS
         2. Verifried & decrypted before using after OS had access
   c. Checking
      i. On add page to page table, check that physical contents are correct for the offset into the object (and HAP has access to object...)

8. Memory Addressing
   a. Goal: verify OS installs correct page
      i. How do it the hard way:
         1. monitor writes to app page table
         2. look at data written
         3. check to see if is correct – look up physical page to see what is in it, virtual page to see what it should contain
   b. InkTag approach:
      i. App creates a list of regions
         1. OIDs, virtual address start & end
      ii. App passes capability (index) to region to kernel
      iii. When kernel handles fault in region, it passes capability to HV with virtual & physical page
      iv. HV verifies it is correct
v. BIG PRINCIPLE: much easier to verify something when it is laid out than to figure out what is happening.
vi. HOW VERIFY NON OVERLAP:
   1. kernel must provide index of previous/next regions – app can verify non-overlap

9. Access control
   a. Goal: want to set permissions on things
      i. Access Control Lists (ACLS)
         1. Discretionary: users can set ACLs on things themselves and decide who gets access
         2. Mandatory: system sets permissions automatically. Think of classified docs: if you have a high security rating, everything you create may be classified.
   b. Objets: things to be protected. Have an ACL made of ACEs
      i. Subject identifier : permission
         1. User Mike : Read
         ii. ACEs can be “ALLOW” – grant acess
         iii. ACESs can be “DENY” – explicitly deny access to some people
            1. hard to use, only in Windows, not shown to users
   c. Subjects: things that can make access – users, groups
      i. Who creates subjects?
         1. In most systems, administrators create users and groups
         2. In inkTag:
            a. admins create users, default groups
            b. users can create sub-users:
               i. user.swift.iPhoto
               ii. Can grant access to subset of files
            c. Users can specify who can use a sub-user:
               i. grant user.swift.iPhoto access to user.Mark
                  1. Now Mark can access things as user.swift.iphone
               ii. grant apps.login access to user.swift
                  1. now apps.login can become user.swift to launch my programs

10. Files & Storage
    a. Goal: store identify of each block with the block
       i. hash & object ID
       ii. Maintain sequential performance (don’t put metadata in a separate partition)
    b. Approach: virtual disk to interleave data & metadata
       i. 4 data blocks + 1 metadata block (5 blocks) look like 4 to the guest
       ii. Need to keep metadata consistent with data on modifications, but cannot write atomically
           1. so keep two versions – old + new – and check both
c. Approach: guest OS identifies object when write block