Scribe Notes 2007-09-04

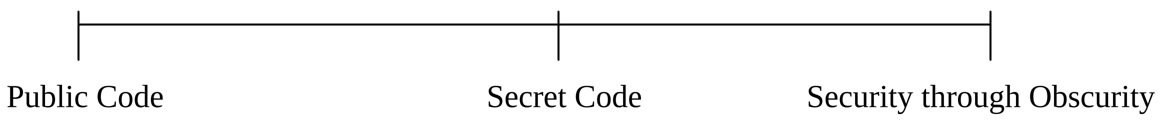
Administrative announcements

- The time of our class will NOT be moved.
- You need to read the 3 articles by Thursday and bring 3 summaries of the articles to class on Thursday.

1 Open design controversy

Should you keep post security critical code so that other people can help you find the security holes?

Each system will have to place itself somewhere on the spectrum from public code to completely secret code.



1.1 Public Code

Post your code publicly online and solicit recommendations.

Pros of Public Code

- Many Eyes – many people can look at your code and help you find the problems
- People write better code if they know it is going too be publicly available.

Cons of Public Code

- An attacker needs to find only one hole in your code to win. You have to fill all of the holes in the code to win.
- The majority of the people inspecting your public code are probably adversaries.
- There are IP issues.

1.2 - Secret Code

Try to keep your code a secret and don’t post it online.

Pros of Secret Code

- Your source code isn’t directly available to attackers

Cons of Secret Code

- You can’t rely on your code really being a secret
    - People could reverse engineer your code
    - Your code could be leaked by an insider
- You don’t get the benefit of “Many Eyes”

1.3- Security through obscurity

Keep the code and design of your system a secret.

Pros of Public Code

- This is in the spirit of keeping as much information from adversaries as possible.

Cons of Public Code

- This isn't practical because you have to assume that the adversary knows the design of your system.

1.4– Summary of open design controversy

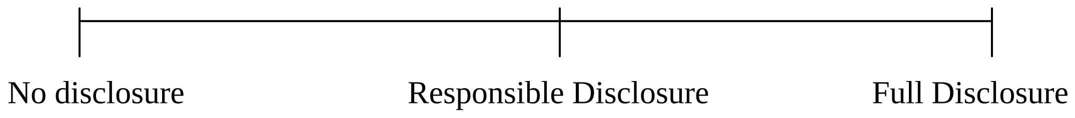
Don't rely on the secrecy of your code for reliability. If you think you'd be in a pickle if your code got leaked, your system isn't secure enough.

Kickoff's Rule **–** The security of your system shouldn't rely on the secrecy of your algorithms, only on the secrecy of your crypto keys. If the security of your crypto keys is compromised, you should be able to change your keys quickly and easily.

2 Publicizing security holes

Who should you tell if you find a security hole in a system?

In each situation a person will have to place themselves somewhere on the spectrum from tell everyone to tell only the vendor.



2.1 No Disclosure

Don't tell anyone but the vendor of the software

Pros of No Disclosure

- You haven't notified potential adversaries of the security hole

Cons of No Disclosure

- Customers are vulnerable
- The vendor might never fix the problem

2.1 - Responsible Disclosure

Tell the vendor of a product about the security vulnerability and threaten to tell the public about it after some fixed amount of time.

Pros of Responsible Disclosure

- Tries to ensure that the vendor will fix the problem
- Doesn't immediately make the vulnerability accessible to potential attackers

Cons of Responsible Disclosure

- Customers are vulnerable and don't know it

2.2 – Full Disclosure

Publicly post the security hole and a work around.

Pros of Full Disclosure

- People are alerted that the system has security vulnerability and can immediately stop trusting the system.

Cons of Full Disclosure

- Adversaries are going to be able to respond much more quickly than the vendor to fix the problem. Even once a patch is made; it takes a long time to get people to apply the patch. During this time user machines are vulnerable.

Full Disclosure Modification

- If you only tell people that there is a problem they can stop trusting the system. Although this maybe be useful enough information to an adversary who would then know where to poke around to exploit the security hole.

3 Three Pieces of Conventional Wisdom

- Be Conservative
  - Whenever there is a possibility that something will go wrong, assume that it will.
  - If there is ever ambiguity about what would happen, assume that it will favor the adversary.
  - Assume that the adversary knows everything about your system and has the source code.
  - Evaluate based upon worst case failure mode.
- Kickoff's rule
  - Don't rely on the security of your algorithm only the security of your crypto keys – which you can change.
- Study Attacks

4 Protection in a multi-user operating system

4.1 - Description of a simple system

We have a computer with no operating system and no security software. This computer is shared by multiple users. (Imagine we're back in an era where computers are very expensive and we need multiple people to share a computer) Assume that Alice uses the computer first and then Bob uses the computer.

4.1.1 – What are some of the security risks that Alice or Bob might encounter?

4.1.1.1 Breaches of confidentiality

- if Alice uses a program that left its secrets around, bob could read whatever was left in system
- Alice could install a key logger that stays running when bob is using the machine

4.1.1.2 Failure of Availability

- Alice could hog all the CPU power so that bob couldn't use it
- bob could accidentally delete all of Alice's files
- Alice could put in a fake login screen so that when bob logs in, Alice would get his login (but if we don't have any security software, we don't have to log in)
- Alice could have left a locking program so that bob couldn't use the computer

4.1.1.3 Failure of Integrity

- Alice could place false information where bob's program would look for it
- Alice could act as bob and send love notes
- Alice could put a back door into the programs on the computer that messes up bob's calculations

4.2 Description of Two "Ideal" Secure Systems

These two examples are ideals. It can be helpful to compare real world systems to these examples to see how close we're getting to a completely secure environment. Each example shows some security properties that would be nice to have in other systems. We wouldn't build a real system these ways - but maybe we want it to offer the same benefits.

4.2.1 Give each person their own computer

If Bob and Alice both have their own computer and the computers are not connected in any way, there is no way for either of them to harm each other.

4.2.2 Have an operator execute the programs in a secure environment

We now have only one computer, but it is operated in a physically secure machine room. Only the operator can gain access to the computer and the secure room. If Alice or Bob want to use the system, they prepare a DVD with their code to give to the machine’s operator to run. The operator will run the DVD and then print out the results and hand them back to Bob or Alice. Before running a second person’s program, the operator will reset and reboot the computer to clear out any state, power down, wipe hard disk and re-install the OS.

Under some reasonable assumptions we can get confidentiality, integrity and availability

Assumptions:

- assume the operator is not malicious and doesn't make mistakes (the operator is TRUSTED - security guarantees only apply if the operator is not malicious and is flawless)
- assume the state of the machine gets completely wiped between runs
- assume Alice and bob use the system at different times (mutual exclusion)
- need verification that the person passing over the program to run is the one who receives the results
- assume physical security (no one can sneak in)
- assume no network on the computer

4.3 How do today’s operating systems do this?

4.3.2 Five minute refresher on how page tables/MMU etc work

$$\begin{tikzcd}\boxed{\textbf{CPU}} \arrow[r] & \boxed{\textbf{MMU}} \arrow[r] & \boxed{\textbf{RAM}}\end{tikzcd}$$

The CPU passes a virtual address to the MMU.
The MMU maps that virtual address to a physical address in RAM.

CPU State
- Mode bit which tells supervisor mode vs. user mode
- interrupt handler device (Address for where the OS is)
- registers

4.3.2.1 Supervisor vs. User Mode

When you're in supervisor mode - you have FULL power (known as ring zero). If you’re not in supervisor mode you’re in USER mode (ring one) User mode is what applications typically run in. There are certain things you can only do in supervisor mode. In supervisor mode - you can CHANGE the mapping in the MMU. A system call or interrupt can be activated in user mode that sets it be supervisor mode, but then it immediately jumps to the interrupt handler address default

4.3.2.2 Booting an application

Loads user application into ram. Set up some mappings in the MMU (only have mappings in place that the application should have access to). The application can't modify kernel state because the user mode is set to "user". Application gets to act like it has ALL of the memory to its self and the MMU handles it using actually only the part of RAM that it has access to. Program 1 can only write to its memory.

4.3.3 **Benefits from the CPU/MMU/RAM system**

Confidentiality:

- I can't read your page b/c I don't have the mappings and only supervisor can write mappings

Integrity:

- No one else can write to my data so my data will have integrity.

Availability:

- Nothing prevents a program from entering an infinite loop
  - A simple solution to that is that the CPU has a timer that counts down to trigger an interrupt - to transition to supervisor mode where then the OS can save the state of the CPU (check point the CPU) - and then restore the check point of some other program so that the other program can run. (in round robin fashion or some other way)

4.3.4 Isolation

The system described so far is completely isolated from every other program. (Not necessarily users)

4.3.4.1 In what ways does this fall short of perfect isolation, like we had with the operator in the secure room?

- One application could allocate all of the ram and not leave anything for anyone else (depending upon how the OS allocated resources) - so availability could be imperfect
- Note this is not very useful
- This approaches the properties of #1 (every user gets their own machine) but it isn't perfect
- If I have a process running on a machine that you're running.
  - I can see if your process is using up any CPU time.
  - I may be able to see how much ram you've got by seeing how much ram I can allocate.
  - one process may be able to gain information about another process
  - Process A might be able to communicate with process B using this timing information
- any time there is timing/resource stuff - you have isolation violation
  - UNLESS you schedule everything without knowing what things need to do, but it completely defeats the time sharing purpose.
  - in practice you don't get perfect isolation

4.4 Controlled Sharing

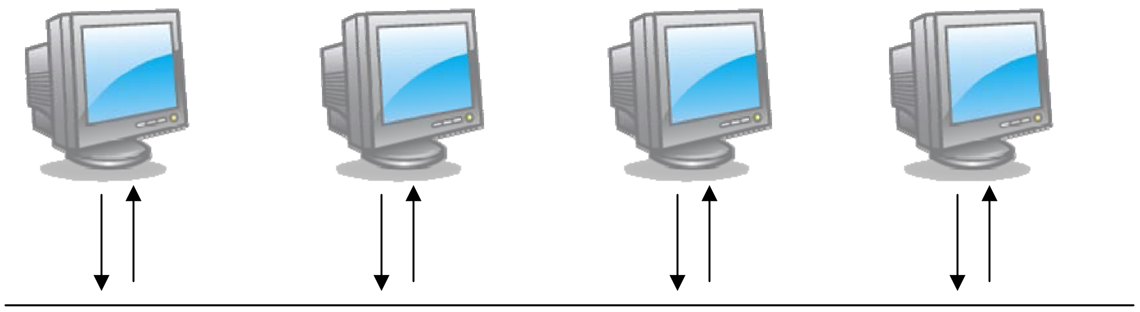
What we really want is controlled sharing. Extend the OS to allow processes to be able to communicate. When process #1 wants to send a message to process #2, it passes that stuff to the OS. Then the OS stuff copies that information into the recipient process. (There are a lot of variations in the specifics of this.)

4.4.1 Butler Lampson

This is an abstract model that illustrates what we're trying for (and later we can look at specific implementations)

- You have a bunch of processes and each one has their own machine
- Each computer/process has their own OS, Ram etc.

"Domain": sometimes referred to as a "protected subsystem", this could be a module. The key thing is that a domain is something with behavior and state. It is an active entity that has some code that one can access and has data that is private. It may have several public entry points. Think of this like a java object: The behavior is whatever is specified by its public methods and it has private instance fields.



- In his model each of these protected subsystems run on their own machine and they are connected by a trusted network (run by some referee that is trusted by everyone)
- The network is how all the systems communicate (without the network none of them can communicate at all)
- We expose a message passing API on the trusted network
- Domain 1 (D1) can send a message to Domain 2 (D2)
    - D1 can say please send X message to D2
    - The network prepends name of sender so that the sender can't forge this "from" field.

4.4.2 How would you implement message passing on a typical OS?

- Send Msg() system call allows some process to send data to another process and then it attaches the id of the sender.
    - Except this doesn't meet the same requirements because process IDs are reused (after a process dies) so you may not know who sent a message

- If you implement send msg() you might try to avoid the need to copy data.
  - The most naive method just copies data from the sender to the kernel, then kernel to the recipient.
  - one optimization is you try to avoid EVER copying the data
    - Map the sender's address space into the recipients address space.
    - that has some issues:
      - if program 2 can also write to the data then program 2 could change the data
        - maybe only give program 2 the read permissions to solve this problem
      - The sender could modify the data AFTER it is sent (the recipient might not be expecting it)
        - Race condition: if the receiver checks the validity of the data and then uses the data, the data could be changed after the validity check
    - You could do a copy on write (if the sender tries to write) Lazy copy.
    - You could require the sender change the permissions on the data to be read only (for the sender)
    - if you're not sending data that is page aligned, you could be leaking other data
  - GARBAGE COLLECTION:
    - The sender and the recipient have references to this shared data. (Whose quota should I charge?)
    - When can I garbage collect this data? (this is a real pain in the butt)
  - There is no solution that doesn't have problems and the easiest way to make this secure is to just always copy the data!

(there was something we left out of the simplistic model - assume all of the components are active and run code - but typically there are some passive items, a keyboard, a printer, a file (the file doesn't have any code))
If we have only active objects we could control that D2 only ever gets information from D1. Passive objects can't check who is passing them messages.
The usual solution is that the OS implements some access control policy - read all - write one, etc.

4.4.3 Access Control Matrices

There are two types of objects:
- Active objects that are sometimes called "Subjects"
  - This could be a process that runs code
- Passive objects that are referred to as “Objects”
  - This could be a file

Control Matrix
- Columns are passive objects

- Rows are Subjects

	Object: /home/daw/foo	Object: /home/bob/bar
Subject: Daw	Read write	Read
Subject: Bob		Read write

At each point in the table you list all of the permissions that User/Subject has for that File/Object.

4.4.3.1 Storage techniques for Control Matrix

- ACL
    - For each column (Object), store each User/Subject that has permissions to that Object and what permissions they have.
- Capabilities Storage
    - For each row (Subject/User), store each Object that it has permission to access and what type of access it has.

4.4.3.2 Control Matrixes in the real world

- In real life - access control matrixes are dynamic
- constantly adding new users and creating new files
- we may change permissions in existing cells

4.4.3.3 All the files that Bob *could* get write permission to:

Ownership: if a User/Subject owns an Object, they can give out access to that Object to anyone else.

- owners can share their permissions
- anyone can drop privileges that they have
- anyone can create new objects and set other people's control
- I can run some program that performs the above operations.

If I want to know what Bob *could* get write permission to...

- what permissions can bob add the matrix
- what permissions can bob convince someone to add
    - You can’t know what permissions bob can convince someone else to add
- This is can't be decided