6.033: Security - Introduction
Lecture 20
Katrina LaCurts, lacurts@mit.edu

```
*********************************************************************
* Disclaimer: This is the beginning of the security section in      *
* 6.033. Only use the information you learn in this portion of the  *
* class to secure your own systems, not to attack others.           *
*********************************************************************
```

0. Intro
   - Previously in 6.033: Building reliable systems in the face of more-or-less random, more-or-less independent failures.
   - Today: Building systems that uphold some goals in the face of targeted attacks from an adversary
   - What can an adversary do?
     - Personal information stolen
     - Phishing attacks
     - Botnets
     - Worms/viruses
     - Etc.

1. Computer security vs. general security
   - Similarities:
     - Compartmentalization (different keys for different things)
     - Log information, audit to detect compromises
     - Use legal system for deterrence
   - Differences
     - Internet = fast, cheap, scalable attacks
     - Number and type of adversaries is huge
     - Adversaries are often anonymous
     - Adversaries have a lot of resources (botnets)
     - Attacks can be automated
     - Users have poor intuition about computer security

2. Difficulties of computer security
   - Aside from everything above..
   - It's difficult to enumerate all threats facing computers
   - Achieving something despite whatever an adversary might do is a negative goal
     - Contrast: an example of a positive goal is "katrina can read grades.txt". Can easily check to see if the goal is met.
     - Example of a negative goal: "katrina cannot read grades.txt". Not enough to just ask katrina if she can read grades.txt and have her respond "no".
       - Hard to reason about all possible ways she could get access: change permissions, read backup copy, intercept network packets..
   - One failure due to an attack might be one too many (e.g., disclosing grades.txt even once)

- Failures due to an attack can be highly correlated; difficult to reason about failure probabilities.
- As a result: we have no complete solution. We'll learn how to model systems in the context of security, and how to assess common risks/combat common attacks.

3. Modeling security
   - Need two things;
     1. Our goals, or our "policy"
        - Common ones:
          - Privacy: limit who can read data
          - Integrity: limit who can write data
          - Availability: ensure that a service keeps operating
     2. Our assumptions, or our "threat model"
        - What are we protecting against? Need plausible assumptions
        - Examples:
          - Assume that the adversary controls some computers or networks but not all of them
          - Assume that then adversary controls some software on computers, but doesn't fully control those machines
          - Assume that the adversary knows some information, such as passwords or encryption keys, but not all of them
   - Many systems are compromised due to incomplete threat models or unrealistic threat models
     - E.g., assume the adversary is outside of the company network/firewall when they're not. Or don't assume that the adversary can do social engineering.
   - Try not to be overambitious with our threat models; makes modularity hard.
   - Instead: be very precise, and then reason about assumptions and solutions. Easier to evolve threat model over time.

4. Guard Model
   - Back to client/server model
   - Usually, client is making a request to access some resource on the server. So we're worried about security at the server.

```
                 Server
Client ----> [ resource ]
```

   - To attempt to secure this resource, server needs to check all accesses to the resource. "Complete mediation"
   - Server will put a "guard" in place to mediate every request for this particular resource. Only way to access the resource is to use the guard.

```
                 Server
Client ----> [ guard | resource ]
```

- Guard often provides:
  - Authentication: verify the identify of the principal. E.g., checking the client's username and password.
  - Authorization: verify whether the principal has access to perform its request on the resource. E.g., by consulting an access control list for a resource.
- Guard model applies lots of places, not just client/server
- Uses a few assumptions:
  - Adversary should not be able to access the server's resources directly.
  - Server properly invokes the guard in all the right places.
  - (We'll talk about what happens if these are violated later)
- Guard model makes it easier to reason about security
- Examples:
  1. UNIX file system
     - client: a process
     - server: OS kernel
     - resource: files, directories
     - client's requests: read(), write() system calls
     - mediation: U/K bit and the system call implementation
     - principal: user ID
     - authentication: kernel keeps track of a user ID for each process
     - authorization: permission bits & owner UID in each file's inode
  2. Web server running on UNIX
     - client: HTTP-speaking computer
     - server: web application (let's say it's written in python)
     - resource: wiki pages (say)
     - requests: read/write wiki pages
     - mediation: server stores data on local disk, accepts only HTTP requests (this requires setting file permissions, etc., and assumes the OS kernel provides complete mediation)
     - principal: username
     - authentication: password
     - authorization: list of usernames that can read/write each wiki page
  3. Firewall. (A firewall is a system that acts as a barrier between a, presumably secure, internal network and the outside world. It keeps untrusted computers from accessing the network.)
     - client: any computer sending packets
     - server: the entire internal network
     - resource: internal servers
     - requests: packets
     - mediation:
       - internal network must not be connected to internet in other ways.
       - no open wifi access points on internal network for adversary to use.

        - no internal computers that might be under control of adversary.
    - principal, authentication: none
    - authorization: check for IP address & port in table of allowed connections.

5. What can go wrong?
   1. Complete mediation is bypassed by software bugs
   2. Complete mediation is bypassed by an adversary
   - How do we prevent these things? Reduce complexity: reduce the number of components that must invoke the guard.
   - In security terminology, this is the "principle of least privilege". Privileged components are "trusted". We limit the number of trusted components in our systems, because if one breaks, it's bad.
   3. Policy vs. mechanism. High-level policy is (ideally) concise and clear. Security mechanisms (e.g., guards) often provide lower-level guarantees.
   4. Interactions between layers, components. Consider this code:
      ```
      > cd /mit/bob/project
      > cat ideas.txt
        Hello world.
        ...
      > mail alice@mit.edu < ideas.txt
      ```
      Seems fine. But suppose in between us cat'ing ideas.txt and mailing Alice, Bob changes ideas.txt to a symlink to grades.txt.
   5. Users make mistakes.
   6. Cost of security. Users may be unwilling to pay cost (e.g., inconvenience) of security measures. Cost of security mechanism should be commensurate with value.