```
Security
-------------------
I. 2 Intro Examples
II. Security Overview
III. Server Security: Offense + Defense
IV. Unix Security + POLP
V. Example: OKWS
VI. How to Build a Website

I. Intro Examples
--------------------
1. Apache + OpenSSL 0.9.6a (CAN 2002-0656)
  - SSL = More security!

  unsigned int j;
  p=(unsigned char *)s->init_buf->data;
  j= *(p++);
  s->session->session_id_length=j;
  memcpy(s->session->session_id,p,j);

 - the result: an Apache worm

2. SparkNotes.com 2000:
 - New profile feature that displays "public" information about users
   but bug that made e-mail addresses "public" by default.
 - New program for getting that data:

  [This link is no longer available because the program has changed.]

II. Security Overview
----------------------

What Is Security?
 - Protecting your system from attack.

 What's an attack?
 - Stealing data
 - Corrupting data
 - Controlling resources
 - DOS

 Why attack?
 - Money
 - Blackmail / extortion
 - Vendetta
 - intellectual curiosity
 - fame

Security is a Big topic

 - Server security -- today's focus.  There's some machine sitting on the
   Internet somewhere, with a certain interface exposed, and attackers
   want to circumvent it.
    - Why should you trust your software?

 - Client security
    - Clients are usually servers, so they have many of the same issues.
```

- Slight simplification: people across the network cannot typically initiate connections.
- Has a "fallible operator":
    - Spyware
    - Drive-by-Downloads

- Client security turns out to be much harder -- GUI considerations, look inside the browser and the applications.
- Systems community can more easily handle server security.
- We think mainly of servers.

III. Server Security: Offense and Defense

- Show picture of a Web site.

Attacks	Defense
1. Break into DB from net	1. FW it off
2. Break into WS on telnet	2. FW it off
3. Buffer overrun in Apache	3. Patch apache / use better lang?
4. Buffer overrun in our code	4. Use better lang / isolate it
5. SQL injection	5. Better escaping / don't interpret code.
6. Data scraping.	6. Use a sparse UID space.
7. PW sniffing	7. ???
8. Fetch /etc/passwd and crack PW	8. Don't expose /etc/passwd
9. Root escalation from apache	9. No setuid programs available to Apache
10. XSS	10. Filter JS and input HTML code.
11. Keystroke recorded on sys-admin's desktop (planetlab)	11. Client security
12. DDOS	12. ???

Summary:
- That we want private data to be available to right people makes this problem hard in the first place. Internet servers are there for a reason.
- Security != "just encrypt your data;" this in fact can sometimes make the problem worse.
- Best to prevent break-ins from happening in the first place.
- If they do happen, want to limit their damage (POLP).
- Security policies are difficult to express / package up neatly.

IV. Design According to POLP (in Unix)

- Assume any piece of a system can be compromised, by either bad programming or malicious attack.
- Try to limit the damage done by such a compromise (along the lines of the 4 attack goals).

<Draw a picture of a server process on Unix, w/ other processes>

What's the goal on Unix?
- Keep processes from communicating that don't have to:
    - limit FS, IPC, signals, ptrace
- Strip away unneeded privilege
    - with respect to network, FS.
- Strip away FS access.

How on Unix?

- setuid/setgid
- system call interposition
- chroot (away from setuid executables, /etc/passwd, /etc/ssh/..)

<show Code snippet>

How do you write chroot'ed programs?

- What about shared libraries?
- /etc/resolv.conf?
- Can chroot'ed programs access the FS at all? What if they need to write to the FS or read from the FS?
- Fd's are *capabilities*; can pass them to chroot'ed services, thereby opening new files on its behalf.
- Unforgeable - can only get them from the kernel via open/socket, etc.

Unix Shortcomings (round 1)

- It's bad to run as root!
- Yet, need root for:
    - chroot
    - setuid/setgid to a lower-privileged user
    - create a new user ID
- Still no guarantee that we've cut off all channels
    - 200 syscalls!
    - Default is to give most/all privileges.
- Can "break out" of chroot jails?
- Can still exploit race conditions in the kernel to escalate privileges.

Sidebar

- setuid / setuid misunderstanding
- root / root misunderstanding
- effective vs. real vs. saved set-user-ID

V. OKWS

- Taking these principles as far as possible.
- C.f. Figure 1 From the paper..
- Discussion of which privileges are in which processes

<Table of how to hack, what you get, etc...>

- Technical details: how to launch a new service
- Within the launcher (running as root):

<on board:>

```
// receive FDs from logger, pubd, demux
fork ();
chroot ("/var/okws/run");
chdir ("/coredumps/51001");
setgid (51001);
setuid (51001);
exec ("login", fds ... );
```

- Note no chroot -- why not?
- Once launched, how does a service get new connections?

- Note the goal - minimum tampering with each other in the case of a compromise.

Shortcoming of Unix (2)

- A lot of plumbing involved with this system. FDs flying everywhere.
- Isolation still not fine enough. If a service gets taken over, can compromise all users of that service.

VI. Reflections on Building Websites

- OKWS interesting "experiment"
- Need for speed; also, good gzip support.
- If you need compiled code, it's a good way to go.
- RPC-like system a must for backend communication
- Connection-pooling for free

Biggest difficulties:

- Finding good C++ programmers.
- Compile times.
- The DB is still always the problem.

Hard to Find good Alternatives

- Python / Perl - you might spend a lot of time writing C code / integrating with lower level languages.
- Have to worry about DB pooling.
- Java -- must viable, and is getting better. Scary you can't peer inside.
- .Net / C#-based system might be the way to go.

==

Extra Material:

Capabilities (From the Eros Paper in SOSP 1999)

- "Unforgeable pair made up of an object ID and a set of authorized operations (an interface) on that object."
  - c.f. Dennis and van Horn. "Programming semantics for multiprogrammed computations," Communications of the ACM 9(3):143-154, Mar 1966.
- Thus:
  <object ID, set of authorized OPs on that object>
- Examples:
  "Process X can write to file at inode Y"
  "Process P can read from file at inode Z"
- Familiar example: Unix file descriptors

- Why are they secure?
  - Capabilities are "unforgeable"
  - Processes can get them only through authorized interfaces
  - Capabilities are only given to processes authorized to hold them

- How do you get them?
  - From the kernel (e.g., open)
  - From other applications (e.g., FD passing)

- How do you use them?

- read (fd), write(fd).

- How do you revoke them once granted?
  - In Unix, you do not.
  - In some systems, a central authority ("reference monitor") can revoke.

- How do you store them persistently?
  - Can have circular dependencies (unlike an FS).
  - What happens when the system starts up?
  - Revert to checkpointed state.
  - Often capability systems chose a single-level store.

- Capability systems, a historical prospective:
  - KeyKOS, Eros, Cyotos (UP research)
    - Never saw any applications
  - IBM Systems (System 38, later AS/400, later 'i Series')
    - Commercially viable
- Problems:
  - All bets are off when a capability is sent to the wrong place.
  - Firewall analogy?