Software Defense Strategies

Principles of Secure System Design
(See: The protection of information in computer systems by Saltzer + Schroder)
http://www.cs.virginia.edu/~evans/cs551/saltzer/

1. Economy of Mechanism
   - KISS = Keep It Simple Stupid
   - Small TCB = (trusted computer base) / Software and Hardware critical to security
     - Ex: Passwords in a flat file vs. database
   - Fewer Bugs
   - Easier to verify

2. Fail Safe Defaults
   - Default: Deny
   - Take Safest Possible Action
     - Ex: File System ACL’s / Firewall Rules / Error handling
   - Default Install Configuration

   - Fails Loudly vs. Failing Silently
     (If your default deny but someone you want to allow can’t get in they will complain to you, this is failing loudly. If you fail silently your system maybe compromised but everyone who’s supposed to used the system can also use it. ) (Too strict rules = lead to complaints but not security compromise, too relaxed rules leads to nothing......)

3. Least Privilege
   - Each process / system should have least privilege to do its job.
   - Ex: Paper Called CLAMP:

Web application Architecture:

Database, Web Application Server, Browser (Separate Servers)
If someone has access to the Web Application they can control not only the database but the info that the user/browser is accessing as well as possibly compromising the user through their browser.

So have a separate database(or privileged tables) one about users and one about products.

What the paper(Saltzer + Schroder) proposes is:

If an anonymous user connects, you connect to a nobody server, they introduce a new components called a query filter. The query filter only has pre-allowed views or queries that it can do by default regardless of what kind of query the user tries to make.

Another part of the solution is for someone who logs in as a user, let’s say Brijesh, you would have the server spawn a virtual machine for only the views that the Brijesh would be allowed access to.

4. Avoid Redundant Parsing  (not part of original paper)
   - XSS = Content sniffing attacks are possible due to redundant parsing.
5. Least Shard Mechanism
   - Least Shared State (2 subsystems don’t share database between them, keep them separate)
   - Software Diversity (Big deal a few years ago) /
     Ex: Monoculture(everyone using windows) on the Internet means you can make one bug that affects many or all systems

6. Complete Mediation
   - Try to know and/or control all means of accessing a resource. The challenge here is knowing that you’ve covered all ways of accessing said resource.

   Ex: There are about 300 system calls in a Linux OS. There should be a permission check accordingly for all of these. So instead of jumping directly to a system call and then check permissions you can have a permissions check first and then jump to the system call. This way all system calls get passed through this permission check resource. (There can be some problems with this system for complex interfaces however)

   - Data Problem = you don’t want some input to become output without being sanitized.

   **Kernel Ex:**

   USER SPACE:

   int read (int fd, Char * bug, int len);
   -----------------------------

   KERNEL SPACE:

   int sys_read(int fd, char * buf, int len)
   {
     memcpy(buf, tmp, len);
   }

   ^^ Evil Pointer that points to kernel space and not to user space could exist in this situation

   Kernel developers realized there was a problem with bad/evil pointers and so created a function to safely copy data:

   copy_to_user(buf, tmp, len);  (takes buf and verifies that it’s a pointer to user space and not kernel space)

   ^^ the problem is that they don’t have to necessarily use this function, so this is not Complete Mediation.

   ***So don’t necessarily make secure behavior easy, make insecure behavior hard or impossible.***

   **Ex:**

   Struct user_ptr {
     int dummy;
     void * p;
   }

   USER SPACE:

   int read (int fd, Char * bug, int len);
   -----------------------------
KERNEL SPACE:

int sys_read(int fd, struct user_ptr buf, int len) {
  memcpy(buf, tmp, len); }

^so now if you use mem copy you will get an error and realize you must now use

    copy_to_user(buf, tmp, len);

This makes it harder to not use copy_to_user.

    -Get the compiler to help. Include dangerous data as a different type.