Chapter 7

Security

The term computer security covers a number of areas and goals. Most of them fall under the following categories:

- Confidentiality of data. As a user of a computer system, this allows you to prevent others from accessing information which you wish to keep private, such as email or passwords.
- Confidentiality of actions. This lets you prevent others from observing what programs you run and what files or external resources you access.
- Integrity of data. Your data will not be modified or deleted without your permission.
- Integrity of operations. Commands should do what they are supposed to do. For example, when you type ls you should get a directory listing, rather than a script that sends your passwords to a secret website in Russia.
- Availability. A system will not stop running when you need it to be operational.

With the rise of the Internet, security has become a much broader field, much of it related to either networking or the behavior of applications such as web browsers. This chapter will cover operating system features which enable computer security, and which reduce the risks from security flaws in application software; the field of computing security is much larger, however[1].

[1] E.g. see courses such as CS 5770, Software Vulnerabilities and Security, CS 6750, Cryptography and Communication Security, CS 6740, Network Security, or CS 6760, Privacy, Security and Usability

7.1 Protection

Much of security involves protection: deciding whether or not to allow an operation based on a series of rules. The purpose of protection is to ensure the security goals described above, by applying these rules to computing operations, allowing some operations and forbiding others. (This is not sufficient for full security, as seen in the discussion below of software vulnerabilities, but it helps.) These rules are typically based on a simple model, of actors, objects, and actions:

- Actors. These perform the actions. At the lowest level these are almost always processes, but they are typically identified by a text or numeric user ID, which is typically associated with either an actual person or a system service.
- Objects. These are the things which are being protected: usually files or directories, but sometimes processes, special devices, configurations, or other aspects of the system which can be modified.
- Actions. These are performed on objects. The most common actions are read and write, but others can include creating and deleting files, killing a process, or rebooting the system.

The goal of the operating system's protection or *access control* mechanisms is to express and enforce policies which determine whether a particular combination of an actor performing an action on an object is to be allowed or denied.

Identity and Authentication

In a Unix-like operating system, the actual actors are processes, which perform actions by issuing system calls. However specifying rules based on the processes themselves—e.g. process 10 may access file `"/home/pjd"`—will not work, because processes are created dynamically: rules could only be made for processes in existence at that time, and not for ones created in the future.

The solution used in almost all operating systems today is the concept of *user identity*: every process is associated with a user identity (e.g. a user name and ID in Unix) and rules are specified in terms of these identities. In its simplest form each of these identities is a login name associated with an individual person, and rules for that identity are used to permit or restrict access by that person[2]. User identity is inherited through the

[2]Additional identities are typically used for *system accounts*, like the `httpd` user associated with the webserver process on many systems. This allows the same mechanism to be used to grant or restrict access for various system operations.

`fork` system call, so that actions taken by a process either directly or indirectly (through children of that process) are bound by the same rules. Access rules specify actors in terms of these identities, and every process is associated with one of these identities, allowing a fixed mapping from identities to permissions.

In practice this requires a login process, or *authentication*, in which an external user proves that she has the right to take on a certain identity. Thus, the person Jane Smith may be given the right to use the operating system identity named `smith.j`, after *authenticating* that identity to the system by providing the correct password. Authentication is an important part of operating system security, as it forms the link between the higher-level goals of system administration (e.g. Jane is allowed to access *file.txt*, but Joe isn't) and the operating system-level features which implement this control.

These authentication mechanisms are frequently called *factors*; hence the term *multi-factor authentication*, where more than one factor (e.g. password, text message) are used. The Wikipedia entry on "Multi-factor authentication" is a good introduction.

Most authentication mechanisms can be classified as one of three types, based on the type of verification provided by the user:

1. *Something you know*: e.g. a password. This is the most common form of authentication, due to ease of implementation.
2. *Something you have*: like a key to a lock, an RSA SecurID token, etc. More complicated to administer, but more difficult for an adversary to obtain.
3. *Something you are*: often biometric data, such as a fingerprint.

You have undoubtedly used many password-authenticated processes; in addition you may have used other methods such as a SecureID token or fingerprint scanner. There are advantages and disadvantages to each type of authentication; however this class focuses on passwords as they are the most widely used.

Checking passwords

Securely storing and checking passwords is difficult, and various methods have been used over the years. The primary alternatives are as follows.

Unencrypted file: Passwords are stored In an unencrypted file, only accessible to the operating system or a privileged user, and a simple string compare is used to verify a user-entered password. Although in principle this should be secure, the result of any error or failure is catastrophic. (This

issue is particularly problematic since people typically re-use passwords across systems.)

Hashed password file: Passwords are put through a one-way cryptographic hash function[3] before being stored. User-entered passwords are hashed by the same algorithm and compared with the stored password hash; if the two match, then the password is correct. Early UNIX systems used this mechanism, and made the password file publicly-readable, as the same file held other information (e.g. mapping between numeric user ID and text username) used by many unprivileged programs. (e.g. the **ls -l** command would need this mapping to show file ownership.) This was considered fairly secure on slow (and especially non-networked) machines, due to the length of time required to crack an individual password by *brute force*—i.e. trying all possible combinations until the correct password was found. However the discovery of *dictionary attacks* changed this, however.

A dictionary attack is based on the idea that in most cases breaking into *any* account is almost as good as breaking a particular account, and takes advantage of the fact that on shared machines (e.g. **login.ccs.neu.edu**) there are a large number of user accounts, increasing the chance of breaking into one of them. The attack consists of calculating the hash for every word in a dictionary[4], and then comparing this list with the hashes in the password file; if *any* of the accounts has a password in this list, the attack succeeds. (a more sophisticated attack, using pre-computed *rainbow tables*[5] is able to crack individual hashed passwords very quickly, as well.)

Password "salt": This is a variation on hashed passwords—when a password is stored, a random string S is chosen (for unknown reasons called the *salt*) and appended to the password before hashing ; the stored value is then [S, hash(S+password)]. Verifying a password is straightforward: the salt value is read from the password file and appended to the input password before it is hashed and compared. This protects against attacks which require pre-computed tables (e.g. dictionary and rainbow table attacks), as now tables would be needed for every possible salt value. (in early usage this was 12 bits long; in modern systems it is 32 bits or more.)

As machines (especially GPUs) become faster and faster, even this method has become less secure over time, as it is becoming feasible for attackers to evaluate billions of possible passwords per second. Counter-measures include making the hash function slower (e.g. running the password

[3] Typically, the password is used as a key to encrypt a known, constant message.

[4] Typically common words, plus personal names, plus variations on those words such as appending a digit.

[5] see `https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture24.pdf`

```
pjd@cs5600-vbox:~$ ls -l /etc/shadow /etc/passwd
-rw-r--r-- 1 root root 1061 Aug 24 2013 /etc/passwd
-rw-r----- 1 root shadow 867 Aug 24 2013 /etc/shadow
pjd@cs5600-vbox:~$ tail -2 /etc/passwd
student:x:1000:1000:A,,,:/home/student:/bin/bash
pjd:x:1001:1001:Peter Desnoyers,,,:/home/pjd:/bin/bash
pjd@cs5600-vbox:~$ sudo tail -2 /etc/shadow
student:$6$JjiTdyS2$cvbtxgVxMwMI5fL0If5Dc90JRuds9yolCKGHc/52ET1tLwksji/
          SNO5pksqdwACztcvhIyCDRfAt9lrK133WA/:15935:0:99999:7:::
pjd:$6$wz5.BTqz$RXkmlCnbb0aoA7C67zf2zL7FokmdKLoc51MLdn7jcDe/JMHzs7iePBC
          NEy7O7ZGbVFIl4wTEbi5a8yhhQALnd1:15941:0:99999:7:::
```

Listing 7.1: Password and shadow password files in Linux

through the hash function 5000 times), and protecting the password file against public access, just like was done with early plain-text password files. (this is done in Linux— `/etc/passwd` contains username and UID information, and is publicly readable, while `/etc/shadow` contains password hashes and is protected.

Challenge-response: This is another variation on password checking, although it is typically used over a network rather than for direct login. In this case the server must keep the password in clear text; when a client requests authentication, the server sends a *challenge*—a random string—which the client adds to the password before hashing it and sending the result back to the server. In this way an attacker with access to the network is unable to learn the password, and (if the server never repeats challenges) is unable to replay previous responses.

Review Questions

7.1.1. In Unix, a password is used to determine if you have permission to access a file: *True / False*

7.1.2. Because the passwords in a password file are encrypted, it is safe to make the file publicly readable: *True / False*

Centralized authentication - LDAP and Kerberos

Modern computer systems frequently use centralized password administration: for instance, when you log in to a CCIS workstation your password is not checked locally, but rather against a central authentication server. The most common used mechanisms are LDAP and Kerberos, frequently used as part of Microsoft's Active Directory service. LDAP (lightweight directory access protocol) is a general-purpose directory protocol that can store information about people, machines, and just about anything

```
# pjd, people, ccs.neu.edu
dn: uid=pjd,ou=people,dc=ccs,dc=neu,dc=edu
displayName: Peter J. Desnoyers
cn: pjd
loginShell: /bin/bash
uidNumber: 11415
gidNumber: 65100
sn: Desnoyers
homeDirectory: /home/pjd
mail: pjd@ccs.neu.edu
givenName: Peter
 ...
```

Listing 7.2: Typical CCIS LDAP entry

else that a computer might want to name; an example entry is shown in Figure 7.2.

One of the attributes an LDAP entry can have is a password: a client can log in to the LDAP server by specifying this password, which will be checked by the server. A Linux or other system will use an LDAP server for authentication by attempting to login with the credentials supplied by the user; if this succeeds, then the local login is successful and user information (such as shell and home directory) is retrieved from the server[6].

Kerberos is a more general-purpose authentication mechanism that allows a server to supply unforgeable cryptographically-signed tickets. These allow untrusted machines, like personal computers, to securely access network services, such as a file server while only having to authenticate once, to the Kerberos server; after this initial authentication, the Kerberos tickets can be used for authentication without having to request additional passwords from the user.

Review Questions

7.1.1. LDAP is used for:

a) Storing user information on a central server
b) Storing (and checking) user passwords on a central server
c) Both of the above.

[6] In Linux and some other systems this is handled in practice by the PAM (pluggable authentication module) framework, developed at Sun Microsystems, which specifies one or more authentication sources for the system to try for various events such as login.

7.2 Unix Access Control

Basic security in an operating system is performed by access control: the process of determining whether each OS action will be allowed, based on the actor (determined by information like a user ID), the specific object (e.g. a file), and object permissions. The desired operation can be described by an access control matrix, such as this one:

	file1	file2	dir1	file3
user1	-	read	-	read/write
user2	read	read	read/write	read
user3	read	read	-	-
user4	read/write	read/write	-	-

Table 7.1: Simple access matrix for four users, 3 files and 1 directory

To be more specific, the Unix security model has the following parts:

Actors: Users. User identity (and file ownership) is described by two IDs:

1. User id (`uid`)
2. Group id (`gid`)

In addition there are permissions for *world*—i.e. any user.

Objects: Files and directories.

Actions: On files: *read*, *write*, and *execute* (i.e. run as a program). Directories: *list* (as in `ls`), *traverse* - i.e. accessing the file `/a/b/c` requires traversing the directories `/a` and `/a/b`, and *modify* - i.e. creating, deleting, or renaming files (or directories) within that directory. (note that list, traverse, and modify are encoded as read, execute, and write permissions)

Users may belong to more than one group: as an example, user pjd belongs to groups faculty, cs5600 and sssl, as shown here by the id command:

```
pjd@login:~$ id pjd
uid=11415(pjd) gid=65100(faculty) groups=1254(cs5600),1294(sssl),65100(faculty)
```

Listing 7.3: Id command output

Files and directories have an owner and a group, and three sets of permissions: one for the file owner, one for members of its group, and one for world, with the permissions typically encoded in a single string, as shown in Figure 7.1.

Finally, (a) only the owner of a file may change its permissions, and (b) each user may belong to some number of groups. (typically up to 32)

Permissions are interpreted as follows:

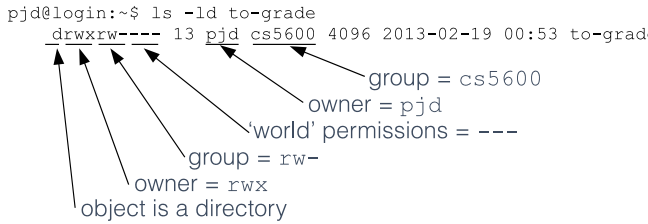


Figure 7.1: Interpreting file ownership and permissions

```
check(process, action, file):
  if process.uid = file.uid:
     if action in file.perm.owner
        allow
     else deny
  if process.gid = file.gid:
     if action in file.perm.group
        allow
     else deny
  if action in file.perm.world:
     allow
  else deny
```

Listing 7.4: File access algorithm

As an example, the access control matrix from earlier: can be encoded in

	file1	file2	dir1	file3
user1	-	read	-	read/write
user2	read	read	read/write	read
user3	read	read	-	-
user4	read/write	read/write	-	-

Table 7.2: Simple access matrix (again)

the set of permissions shown in Figure 7.5.

```
group1 = {user2,user3,user4}
file1: owner = user4, group = group1
               permissions = {owner = 'rw-', group = 'r--', other = '---'}
file2: owner = user4, group = [doesn't matter]
       permissions = {owner = 'rw-', group = 'r--', other = 'r--'}
dir1:  owner = user2, group = [doesn't matter]
               permissions = {owner = 'rw-', group = '---', other = '---'}
group2 = {user1,user2}
file3: owner = user1, group = group2
               permissions = {owner = 'rw-', group = 'r--', other = '---'}
```

Listing 7.5: Permissions for access matrix in Table 7.2

Limitations of Unix permissions

However if we make minor changes to this access control list:

	file1	file2	dir1	file3
user1	—	r–	—	rw-
user2	r–	r–	rw–	r–
user3	rw-	r–	—	rwx
user4	rw-	rw-	—	—

Table 7.3: Complex access matrix (`ls -l` notation used for conciseness)

There are two problems here:

`file1`: Here two users have the highest level of privilege. If `user3` and `user4` are assigned to the same group, and the `file1` owner and group permission are both set to `rw-`, then the only permission left is "world". If that is set to `--` then `user2` will not have the read access specified in the access control matrix; however if it is set to `r-` then `user1` will improperly have access[7].

`file3`: Here there are 4 distinct combinations of permissions, while Unix permissions for a single file can only hold 3 combinations (owner, group, and world).

Review Questions

7.2.1. Given the following file permissions:

```
pjd@login:  ls -l file.txt
-rwxrw-r- 13 pjd faculty 1280 2013-10-19 00:01 file.txt
```

(A) which users can read the file? (B) Which users can write to the file? (C) which users can execute the file?

[7] Although it's possible to achieve this matrix with owner=[`user2 r-`], group=[`(user2,user3,user4) rw-`], and world=[`--`], it doesn't really make sense since the owner can gain write access just by changing permissions on the file.

a) Only user `pjd`
b) Only user `pjd` and any other user in group `faculty`
c) All users

7.2.2. In the following access control matrix:

	file1	file2	dir1	file3
user1	- w -	- - -	r - -	rw -
user2	r - -	r - -	rw -	r - -
user3	r - -	r - -	- - -	r - -
user4	rw -	rw -	rwx	- - -

which of the desired access for which files or directories cannot be implemented using simple Unix permissions?

a) file1 and dir1
b) file1 and file4
c) None: the entire access matrix can be expressed in Unix permissions
d) dir1

Access Control Lists

Access Control Lists (ACLs) are explicit rules granting or denying access to users, and are more powerful than simple permissions. The idea is straightforward: an ACL is a list of rules, where each rule specifies a user or group, an action, and whether to allow or deny permissions.

Using the same example, which could not be encoded in standard Unix permissions:

```
        file1  file2  dir1   file3
user1   ---    r--    ---    rw-
user2   r--    r--    rw-    r--
user3   rw-    r--    ---    rwx
user4   rw-    rw-    ---    ---
```

the desired access to `file1` can be expressed as:

```
file1:      owner = user4, group = {user4,user3}
            owner: rw-, group: rw-, user2: r--, other: ---
file3:      owner = user3, group = {user3, user1}
            owner: rwx, group: rw-, user2: r--, other: ---
```

Access Control List Examples

This example uses OSX access control lists; however, Linux and Windows have similar mechanisms. We start with three user IDs: `pjd`, `guest`, and `joe`.

First a file is created, made readable by all users, and an ACL rule is added denying access to user joe (using the chmod —a command).

```
pjd$ echo 'file 1 contents' > file.1
pjd$ chmod u=r,g=r,o=r file.1
pjd$ chmod +a 'joe deny read' file.1
pjd$ ls -le file.1
-r--r--r--+ 1 pjd wheel 16 Aug 28 14:20 file.1
 0: user:joe deny read
```

The file can now be read by both `pjd` and `guest`, but not `joe`:

```
pjd$ cat file.1
file 1 contents
joe$ cat file.1
cat: file.1: Permission denied
guest$ cat file.1
file 1 contents
```

Now we create a second file, set it owner read-only, and a rule is added giving read access to `joe` but no other user:

```
pjd$ echo 'file 2 contents' > file.2
pjd$ chmod u=r,g=,o= file.2
pjd$ ls -le file.2
-r--------+ 1 pjd wheel 16 Aug 28 14:20 file.2
 0: user:joe allow read
```

Now the file can be read by `pjd` and `joe` but not `guest`:

```
pjd$ cat file.2
file 2 contents
joe$ cat file.2
file 2 contents
guest$ cat file.2
cat: file.2: Permission denied
```

Other Privileged Operations

Most Linux security checking is handled by a combination of the following rules:

- File system permissions
- Signal permissions. A non-root process can only signal processes with the same user ID.
- Super-user. User ID 0 (traditionally named "root") may access any file or signal any process; dangerous kernel operations can only be performed if the current user id is 0.

- Set UID. This allows users to invoke functions with elevated privileges when appropriate.

Most security checking in Linux / Unix is handled by file system permissions. As an example, system utilities need direct access to the disk (e.g. to format a new file system); normal users are prevented from reformatting the disk by the permissions on the special file representing the disk device (e.g. `/dev/sd0`)

Unix signals are primarily used to kill a process, and so are only allowed between processes with the same user ID.

User ID number 0 (traditionally given the username "root") is allowed to bypass all user id-based permissions. In addition, certain system calls (e.g. mount a file system, reboot, install a kernel module) may only be performed by the super-user.

Finally, the setuid permission flag on a file tells the kernel that when the file is executed it should take on the identity of the file's owner, not the user who invoked it. This is a simple mechanism which allows programs to make finer distinctions than the kernel does. For instance the `mount` program is owned by user "root", and marked *setuid*, as under certain circumstances a normal user may be allowed to mount a filesystem (e.g. when it is a removable drive). When the program is run by a normal user, it checks configuration files to see whether the request is authorized; if so it is able to invoke the mount system call as user ID root.

7.3 SELinux

An alternate security mechanism available in Linux is called SELinux, or Security-Enhanced Linux. This is an enhancement to the normal Linux security model, which allows for exceptions to normal Linux rules. As an example, normal file permissions can still deny access to a file, but in cases where permissions allow access, SELinux rules might still forbid it.

SELinux is based on rules about *domains* and *types*. A domain is an execution environment that users run programs in; a set of rules for a domain determine which users can run what programs within that domain. Files have types, and rules determine which domains are able to access which types of files. Finally, users can change domains by running certain programs; when this occurs is again determined by (unsurprisingly) another set of rules.

Rules are loaded into the kernel by a user-space policy process, and file types are determined by an SELinux context associated with the file, stored

in file *extended attributes* in the file inode.

As an example, the password file `/etc/passwd` contains usernames, groups, home directories, but not the hashed passwords themselves, which are in the shadow file, `/etc/shadow`; the shadow file may be modified when users run the `/usr/bin/passwd` program:

```
[root@localhost ~]# ls -Z /etc/passwd /etc/shadow /usr/bin/passwd
-rw-r--r-- root  root  system_u:object_r:etc_t          /etc/passwd
-r-------- root  root  system_u:object_r:shadow_t   /etc/shadow
-r-s--x--x root  root  system_u:object_r:passwd_exec_t /usr/bin/passwd
```

One SELinux policy rule states that a user enters the `passwd_t` domain when executing a file of type `passwd_exec_t`; another states that processes running in the `passwd_t` domain have read and write privileges to files of type `shadow_t`. If SELinux is enabled, then even the superuser will only be able to modify `/etc/shadow` (and update your password) by executing `/usr/bin/passwd` or another executable marked `passwd_exec_t`.

Actual SELinux policies are extremely complex, containing hundreds of rules; if you are interested in finding out more there are a number of tutorials on the Internet, including `http://www.centos.org/docs/5/html/Deployment_Guide-en-US/ch-selinux.html`

Control of Information Flow

File access control is (somewhat) straightforward for an operating system to provide, as it represents a simple decision. If access is allowed, then the requested operation proceeds without interference; if it is denied, then the request fails completely.

In many cases, however, the desired restrictions are more subtle. Perhaps the earliest published example was a simple computer guessing game; the program would need to access the data to be guessed, while preventing the user from accessing it directly. Simple file permissions would not work, as if the game program were able to access the data file, then other programs (e.g. an editor) would be able to as well, allowing users to cheat.

In general such a problem requires interposing higher-layer software between the user and the protected information. In Unix the *setuid* mechanism allows a user A to run a program as a different user B (e.g. root), by having the executable file owned by user B and setting the *setuid* permission on the file. This can be done in a way that user B has full access to the protected data, allowing the program to access the data on behalf of user A, but only in ways allowed by the program logic.

In modern systems it is more common for such a gatekeeper role to be played by a server with access to the data, which accepts requests from users via messages and makes decisions on what data to reveal. Some examples:

MySQL: This database accepts connections over a TCP socket; users log in and then are able to read and modify those tables they have been given permission to access, regardless of which files the data resides in. The MySQL process runs under a separate user ID, which is the only one able to access the underlying data.

Blackboard: Connections to Blackboard are web sessions, controlled directly by users, and isolation of the data itself is ensured by preventing user access to the system that Blackboard runs on. The application logic enforces a complex set of rules governing what information each user may access; thus an instructor may see all grades, while a student may only see aggregate information (e.g. averages) about other students' grades.

Review Questions

7.3.1. Which of the following most accurately describes the effect of the *setuid* attribute on a file?

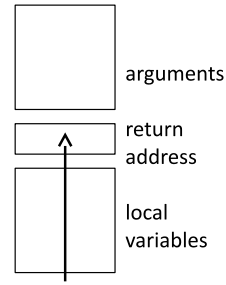
a) It causes the file owner (i.e., user ID) to be updated whenever a process accesses the file
b) It causes the file owner to be updated whenever a process executes the file
c) It causes the user ID of the process to be updated when a process executes the file

7.3.2. Which of the following statements are true? A server-based database such as MySQL:

a) protects the security of its data by putting it in files owned by a separate user ID
b) uses file permissions to prevent access to its data
c) uses application logic on a per-request basis to determine whether to provide access to a data item.

7.4 Attacks — Stack Overflow

In Figure 7.2 you can see a fragment of code that was attacked (among others) by the first piece of Internet malware, the 1988 Morris worm. The target program (`fingerd`) was run with a network connection for its standard input, and used the `gets` function to read a line of input into a buffer on the stack; it would normally return a simple reply based on that input and then exit. `Gets` reads a line from standard input, reading as many bytes as it takes before it finds a newline or reaches end-of-file. The buffer used, located at a lower address on the stack than the return address, was 512 bytes long, but the worm sent a 537-byte single-line message consisting of machine code, ending with a carefully chosen return address pointing at the beginning of the injected code. Since `fingerd` was run as the root user, the result was that when the function returned, the malware code had full control of the machine. In the years since, many lessons have been learned about preventing this sort of attack:




```
f() {
  char buf[512];
  gets(buf);
  ...
}
```

Figure 7.2: Stack frame and buffer overflow

- (application writers) Do not use `gets` or other functions which can overrun a fixed-length buffer. (e.g. `fgets` takes the buffer length as an argument)
- (OS writers) Randomize the location of the stack and libraries each time a program is run, to make it more difficult to guess where an attack should return to.
- (OS writers) Use the NX ("no execute") page table bit on modern processors to prevent code on the stack from being executed.

Unfortunately buffer overflow vulnerabilities are still alive and well, as more sophisticated attacks have been developed to counter these techniques, as long as there is an initial application bug to give access to the stack.[8]

[8] If you are interested in learning more about stack overflows, there is a good tutorial at `http://www.tenouk.com/cncplusplusbufferoverflow.html`

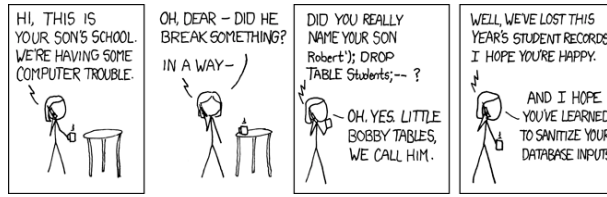


Figure 7.3: Image by Randall Munroe, from Xkcd: http://xkcd.com/327/, licensed under creative commons non-commercial license 2.5

Security - SQL Injection

What happened to Bobby's school's student database? Let's assume they used code like this Visual Basic fragment, adapted from an example on an MSDN discussion board. ('&' concatenates strings in VB):

```
cmd.commandText = "INSERT INTO Students (Name) VALUES ('" \& studentName \& "');"
cmd.executeNonQuery()
```

So if studentName="Joey Smith", the following SQL command will be executed:

```
''INSERT INTO Students (Name) VALUES('Joey Smith');"
```

But if studentName="Robert'); DROP TABLE Students;–", we get:

```
''INSERT INTO Students (Name) VALUES('Robert'); DROP TABLE Students;--);''
```

Semicolon (";") is the command separator in SQL, and "" is a comment marker causing the rest of the line to be ignored; after adding 'Robert' to the Students table, the DROP TABLE command will be executed, removing the entire table.

SSL and Connection Security

Secure Sockets Layer (SSL) allows two systems to establish a connection that cannot be intercepted, even by an adversary who observes every packet sent by both systems. Most importantly, it does not require any shared encryption key to be used by both systems[9]. SSL relies on a combination of private- and public-key encryption:

- Private-key encryption uses a private key to encrypt a message, which may then be decrypted using the same private key.

[9] In most cases using a shared private key doesn't solve the problem: before you use it, you have to figure out how to securely communicate the private key.

- Public-key encryption uses two keys: (a) A public key to encrypt the message, and (b) a separate private key which must be used to decrypt it. In one of today's public-key systems, the public key is the product of two large prime numbers, and the private key is the two numbers themselves. The private key can be derived from the public key by factoring it, but for large numbers this is believed to be prohibitively difficult to actually do.

The simplest use of public-key encryption to provide a private connection would be for the two systems to each have public/private key pairs, send each other their public keys, and then encrypt traffic with the other system's public key. Unfortunately, public-key encryption is very computationally expensive, so instead SSL uses the following steps, sometimes called the SSL handshake:

1. The client connects to the server
2. The server sends its public key to the client
3. The client chooses a random number, encrypts it with the public key, and sends it to the server, which then decrypts it.

Client and server both use this random number as the key to a private-key code — all outgoing messages are encrypted using this key, and all incoming messages are decrypted with it. There are additional aspects of the SSL protocol to guard against impersonation and "man-in-the-middle" attacks, which are somewhat more complicated and are not covered here.

Answers to Review Questions

7.1.1 False. The password authorizes you to log in as a specific user ID; file permissions determine whether that user ID has access to a particular file.
7.1.2 False. This used to be the case, but modern hardware can crack encrypted passwords very quickly.
7.1.1 (3) - LDAP handles both user information and passwords.
7.2.1 Read: (3), all users. ("world" permissions are `r-`) Write: (2), owner and group members. Execute: (1), owner `pjd` only.
7.2.2 dir1 has four separate sets of access, which cannot be encoded in three sets of permissions.
7.3.1 (3) the process ID is set to that of the owner of the file
7.3.2 All of these statements are true.