Why are your linux files secure?

Part 1: the concept of identity.
   user, group, ....
Part 2: the concept of protection.
   the protection word: rwxr-xr--
Part 3: mutating protections (legally).
   chmod, chgrp (chown)
Part 4: mutating protections (illegally).
   today.
What you don't want to see in your account:
  foo mode=rwsrwsrwx owner=root group=root
  Setuid root
  Setgid root
  world executable
  Whoever runs this gets root on your workstation.

What a rootkit actually does, in some form:
  cp /bin/csh foo
  chown root foo
  chgrp root foo
  chmod 06777 foo
  so that anyone who runs foo gets a root shell.

Almost as bad: for a normal user:
  cp /bin/csh bar
  chmod 04755 bar
  so that anyone who runs bar gets access to your account.
  This just requires your privilege, not root.
The chain of privilege

People

login as users,
which have groups and own files,
which determine what the user can do.

Privilege can only be changed via processes.
which can run as invoker (default)
or the owner (or group) (setuid, setgid).
Many people thought that one's privileges are related to parent-child relationships. This is false. The owner of a process retains powers over it, even if it becomes a child of init.

The process has
  the concept of userid (which determines who can send signals to it)
  and the concept of groupid (which determines which group owns files that it creates).
It is **ownership of the process** that determines who can manipulate it.
How login works:

login is run by root.
accepts your password.
hashes it.
compares with (protected) hash of real password (in /etc/shadow or similar).

If there is a match, then:

fork a child
downgrade child privilege to that of user.
excl the user's shell from the child.
Security and Privacy

**Security**: you control the things you own.

**Privacy**: it is not seen by others without your permission.

In an OS, these are (nearly) equivalent concepts.

Two other important concepts

**Identity**: the extent to which the actions of an electronic entity can be traced to a real person.

**Transparency**: the extent to which we can see what people do.

Basic problem of security: the internet exhibits **weak identity** and **low transparency**.
Two levels of security/privacy in an operating system:

**Account security**: can others control your (personal) account?

**OS security**: can others control your operating system itself?

Both of these involve giving a process permissions that it does not deserve.
What can a regular user do?

Grant anyone the power to become the user.
Set traps for other users.
Set up a rogue network service.
Attack other machines (in the local network).
Poison DNS, ARP, etc.
Send email containing malicious code to others.
What can root do?

Compromise the accounts of all users.
Become any user and do anything.
Corrupt services used by external machines.
Poison the basic infrastructure of the internet.
Warning: the law and cracking

The things I will now describe are **illegal**.
I do not condone these things.
You should not engage in them.
But you must know what others can do to you.
Protecting yourself

Assume nothing.
If it looks too good to be true, it is.
Understand that people can do anything,
Even very implausible things.
An aside: human factors of security

According to a 2007 study,

"Do you want to allow this software to make changes?"

is -- in the minds of most users -- semantically equivalent to

"Do you want to get your work done, or not?"

Weakest link in any system is the human user.
Social Engineering

A professional hacker's most effective tool.

Convince others that you are **someone you're not:**

- A microsoft technical support specialist.
- An amazon technical support person.
- An ebay technical support person.

(Use your imagination)

In the natural course of playing that role, **trick your victims** into compromising their security and privacy:

- By running untrusted software.
- By disclosing their passwords, either directly or indirectly.
Security Myths and Realities

Myth:
The main cause of security breaches is **technical**. The most effective attacks **trick the machine**. The most effective attack vector is a **software weakness**.

Reality:
The main cause of very serious security breaches is **human error**. The most effective attacks **trick users**. The most effective attack vector is to **con a user into believing something that isn't true**.

**Cracking is a confidence game.**
Hacking versus cracking

- **Hacking** has a positive and a negative connotation:
  - Positive: someone who writes mystically powerful code.
  - Negative: someone who breaks into computers.
- **Cracking** is less confusing:
  - No positive connotation.
  - Breaks into computers, by any means necessary.
  - Not even necessarily a good programmer.
Why crackers crack?

Why do crackers crack?
   Illusion of power.
   Curiosity.
   Hatred.
   Money.

What do crackers want?
   Anonymity.
      to send Spam
      to attack other machines.
   CPU cycles.
      to run bots and other interesting software.
   Bandwidth.
      to run bots and other interesting software.
Principles of effective cracking

1. **Don't get caught.**

2. In pursuit of (1), **obfuscate your identity** as much as possible.
   a. By using exploited systems to exploit others.
   b. By using exploits that do not require human interaction.

3. Exploit the weakest links:
   a. **Poorly configured systems** (and networks) that aren't adequately protected.
   b. **Naïve users** who will believe what you tell them.
   c. **Previously exploited systems.**
Anonymity: ensuring that your own physical identity is safe.
Leapfrogging: using a previously exploited system to break into another.
Patience: waiting (perhaps) a long time for results.
Curation: filing lists of exploits, systems that can be exploited, and previously exploited systems for future use.
Scanning and fingerprinting: determining lists of systems on which an exploit is likely to work.
Poisoning: utilizing one exploit to set a (more deadly) trap.
Track covering: hiding what you have done to a system.
Hiding identity

• Your identity is as secure as the number of people who have to collaborate to discover it.
• Especially if those people don't get along.
• International discord is a cracker's best friend.

• Ideal scenario
  • Break into a system in South Korea.
  • From there, break into a system in North Korea.
  • From there, break into a system in the United States.
• I call this the "principle of social distance".
Most sophisticated attack vector: **reverse social engineering**

Research what software is used in the site. Fake a major distribution of the software; when installed, it breaks the system. Publish the real phone number of the vendor on the fake. Crack the victim's phone system (a weak point) and redirect the real vendor phone number to you. Send the distribution to the victim with the authentic phone number. So, victim installs software, calls vendor, gets you. At this point, you are in a **near-perfect cracking situation**

Victim will swear that you're official; they have "proof"; they called the official number themselves! You are -- in fact -- a slimeball intent on stealing their secrets.
Technical cracking

- Not the big issue in cracking.
- But a constant nuisance, because a bored 14-year-old can exploit a technical problem.
- And there are lots and lots of bored 14-year-olds!
Three ingredients

Set of exploits
Set of fingerprinted machines (they have the capabilities required by the exploit).
Tools for computing fingerprints.
Previously exploited machines.
Kinds of technical cracking

Tuesday, December 06, 2011   12:03 PM

Prevalent kinds of technical cracking
   Bug exploits.
   Dictionary attacks.
   Poisoning.
What is a (technical) vulnerability?

- Unpredictable user input provides unintentional access.
- Typical example: buffer overflow attack.
- Cause: programming that doesn't protect against arbitrary inputs.

These vary in severity

- programs run by normal users: compromise the user account.
- programs run by root: compromise the system itself.
Buffer overflow attacks

Recall that each stack frame contains
Local variables.
Return addresses.
What you might not realize is that:
You can overwrite these in upper (earlier)
frames from a mistake in a lower (later)
frame.
You can point return addresses at arbitrary
code!
gets is the root of all evil
gets(buf): reads to end of line, potentially past end of buffer.
fgets(buf,len,stdin): reads to either end of line or end of buffer.

So, why don't we just rewrite everything?
Footprint: millions of lines of code.
Subtlety: it is possible to embed a gets-like behavior in other code.
Two basic exploits
  stack buffer overflow: points the return value of a frame at unknown machine code.
  heap buffer overflow: corrupts a pointer to function to point to unknown code.

Super specific:
  tailored to application, os, and version. dependent upon stack layout, buffer size, and many other things.

Finger printing: matches super-specific hacks to super-specific systems.
stack buffer overflow attack

Find an input statement without buffer limit. Engineer machine code to put into the buffer. Make input long enough to overwrite the stack frame return pointer (in your frame, or in the previous (upper) one). So that on return, it executes the code you wrote rather than returning!

Exploit code already in memory to pursue your attack.
Buffer overflow attack vectors

User programs.
Web sites.
Network services (run as root -- yum!)
Properties of an overflow attack

**Application-specific:** only one application.

**Build-specific:** depends on structure of compiled program.

Doesn't work if used on any other application or any other build of the same application. (Can break things).

Fingerprinting is important: it determines where an exploit will work.

**Fingerprinting:**

Using a program to determine the specific configuration of a system, including installed programs and versions.
Systems tend to converge -- on a worldwide scale -- to a single build, so that being able to attack one of them makes one able to attack all of them. -- Dan Geer
Exploiting the systems monoculture

Stack buffer overflow attacks exploit a systems monoculture:
Everyone installs the exact same application program.
If the program has a vulnerability, everyone gets that vulnerability.
Countermeasures for buffer overflow attacks
Frequent patching for discovered overflow vulnerabilities.
System fingerprinting to detect changed files.
System scanning for compromised (setuid) files.
Doctor, it hurts if...
Then don't do that!

It isn't that simple.
Millions of lines of contributed code.
Indirect buffer overflows: lack of protection occurs in an unpredictable place.
Best defense against buffer overflow: fuzz testing
   Generate random inputs.
   Classify behaviors.
   Identify potential overflow behaviors.
Heap buffer overflows:

A program writes into a heap buffer. Input that is too long overwrites other structures. Including structures containing, e.g., function pointers. Result: when function pointer is invoked, the wrong function is called.
Poisoning
   Use one successful exploit as the basis for another. Set up an environment in which users do one thing and have another happen.
Three examples:
   Path poisoning
   DNS poisoning
   ARP poisoning
Path Poisoning

Get users to execute a malicious program by putting it into their execution paths.
It is named like a regular program: a "trojan horse". Example: "ls" that isn't the real "ls".

Malicious program
compromises the account of the user.
potentially installs still more malicious information.
Paydirt: if root executes a trojan, system is compromised.

You might have noticed that '.' is not in your execution path.
This is why you have to execute ./a.out rather than a.out.
This is also why a malicious person cannot fool you into executing his own ./ls.
DNS Poisoning

DNS = Domain Name Service = the service that maps from internet names to internet numbers.
Fool a system into calling up a malicious system instead of google.com.
Malicious system tries to install malware.
You think it's google, so you press "OK".
ARP Poisoning

ARP = address resolution protocol = protocol that maps from internet addresses to local (ethernet) addresses.
In the local network, you contact system X and get system Y.
Often used to accomplish DNS poisoning by pointing at a local rogue DNS server that the cracker set up.
Why is poisoning possible?
The internet was designed as a "trusting" system. There is no provision for establishing authenticity of a network entity. Further, this naive trust is necessary for proper network operation -- more on this in COMP112.