From: Manifestation
Subject: Security holes manifest themselves in (broadly) four ways...
Date: 11.10.93

(Please contribute by sending E-Mail to <scott@santafe.edu> ...)

[quoting from the comp.security.unix FAQ]
Security holes manifest themselves in (broadly) four ways:

1) Physical Security Holes.

- Where the potential problem is caused by giving unauthorised persons physical access to the machine, where this might allow them to perform things that they shouldn't be able to do.

A good example of this would be a public workstation room where it would be trivial for a user to reboot a machine into single-user mode and muck around with the workstation filestore, if precautions are not taken.

Another example of this is the need to restrict access to confidential backup tapes, which may (otherwise) be read by any user with access to the tapes and a tape drive, whether they are meant to have permission or not.

2) Software Security Holes

- Where the problem is caused by badly written items of "privledged" software (daemons, cronjobs) which can be compromised into doing things which they shouldn't oughta.

The most famous example of this is the "sendmail debug" hole (see bibliography) which would enable a cracker to bootstrap a "root" shell. This could be used to delete your filestore, create a new account, copy your password file, anything.

(Contrary to popular opinion, crack attacks via sendmail were not just restricted to the infamous "Internet Worm" - any cracker could do this by using "telnet" to port 25 on the target machine. The story behind a similar hole (this time in the EMACS "move-mail" software) is described in [Stoll].)

New holes like this appear all the time, and your best hopes are to:

  a: try to structure your system so that as little software as possible
  runs with root/daemon/bin privileges, and that which does is known to
  be robust.

  b: subscribe to a mailing list which can get details of problems
  and/or fixes out to you as quickly as possible, and then ACT when you
  receive information.

>From: Wes Morgan <morgan@edu.uky.ms>
>
> c: When installing/upgrading a given system, try to install/enable only
> those software packages for which you have an immediate or foreseeable
> need. Many packages include daemons or utilities which can reveal
> information to outsiders. For instance, AT&T System V Unix' accounting
> package includes acctcom(1), which will (by default) allow any user to
> review the daily accounting data for any other user. Many TCP/IP packa-
> ges automatically install/run programs such as rwhod, fingerd, and
> <occasionally> tftpd, all of which can present security problems.
>
> Careful system administration is the solution. Most of these programs
> are initialized/started at boot time; you may wish to modify your boot

> scripts (usually in the /etc, /etc/rc, /etc/rcX.d directories) to pre-
> vent their execution. You may wish to remove some utilities completely.
> For some utilities, a simple chmod(1) can prevent access from unauthorized
> users.
>
> In summary, DON'T TRUST INSTALLATION SCRIPTS/PROGRAMS! Such facilities
> tend to install/run everything in the package without asking you. Most
> installation documentation includes lists of "the programs included in
> this package"; be sure to review it.

3) Incompatible Usage Security Holes

- Where, through lack of experience, or no fault of his/her own, the
System Manager assembles a combination of hardware and software which
when used as a system is seriously flawed from a security point of view.
It is the incompatibility of trying to do two unconnected but useful
things which creates the security hole.

Problems like this are a pain to find once a system is set up and
running, so it is better to build your system with them in mind. It's
never too late to have a rethink, though.

Some examples are detailed below; let's not go into them here, it would
only spoil the surprise.

4) Choosing a suitable security philosophy and maintaining it.

>From: Gene Spafford <spaf@cs.purdue.edu>
>The fourth kind of security problem is one of perception and
>understanding. Perfect software, protected hardware, and compatible
>components don't work unless you have selected an appropriate security
>policy and turned on the parts of your system that enforce it. Having
>the best password mechanism in the world is worthless if your users
>think that their login name backwards is a good password! Security is
>relative to a policy (or set of policies) and the operation of a system
>in conformance with that policy.

From: Hacking
Subject: Hacking Ideas
Date: 11/10/93

(Please contribute by sending E-Mail to <scott@santafe.edu> ...)

[Many ideas taken from: HaxNet - APG V1.3 : Guide to finding new holes]

NOTE: I think this should be divided into general categories:
1) General principles
2) Looking for holes in src (most items here)
3) Looking in binary distributions
4) Looking in site specific configurations

  The following general classifications suggest themselves:
1) SUID/SGID
2) Return codes/error conditions
3) unexpected input
4) race conditions
5) authentication
6) implicit trust
7) parameters
8) permissions
9) interrupts
10) I/O

```
11) symbolic links
12) Daemons, particularly those taking user input.
13) Kernel race conditions
14) what else? - please add categories

(Suggested splitting of above into main and sub-catagories)
I:   Suid binaries and scripts
     unexpected user interactions
     flawed liberary calls
     implicit assumptions of external conditions (sym links, loc. paths)
     race conditions
II:  daemons running with priviliged uid's
     race conditions
     poor file protectons
     implicit file protections
     trust
     authentication
III: Kernel problems
     Kernel race conditions
     device driver code
```

The following four step method was created by System Development
Corporation, who report a 65% success rate on the flaw hypotheses
generated. Doing a comprehensive search for operating system flaws
requires four steps:

```
Step 1) Knowledge of system control structure.
===============================================
```

  To find security holes, and identifying design weaknesses it is
necessary to understand the system control structure, and layers.
  One should be able to list the:

```
A) security objects: items to be protected. ie: a users file.
B) control objects: items that protect security objects. ie: a i-node
C) mutual objects  : objects in both classes. ie: the password file
```

  With such a list, it is possible to graphically represent a control
hierarchy and identify potential points of attack. Making flow charts
to give a visual breakdown of relationships definitely helps.
  Reading the various users, operators, and administrators manuals should
provide this information.
(following para's should probably be moved to a "legal" section)
  Reading and greping source code should also prove valuable. For those
without a source licence, I would suggest we use LINUX, NET2, and BSD386
distributions in order to stay legal. At some future time we may be able
to form a working contract between someone or a company with legal access
to other distributions and members actively participating in this project.
  It appears that extracts of proprietary code may be used for academic
study, so long as they are not reused in a commercial product - more
checking is necessary though.

```
Step 2) Generate an inventory of suspected flaws. (i.e. flaw hypotheses)
========================================================================
```

In particular we want:
  Code history:
    What UNIX src does a particular flavor derive from? This is important
for cross references (very often only one vendor patches certain code,
which may get reused, in it's unpatched reincarnation by others)
  A solid cross reference:
    Who checked which bug in what OS and what version prevents us from
duplicating work.

  A good start would be listing all the suid binaries on the various OS
flavors/versions. Then try to work out why each program is suid. i.e.:
    rcp is suid root because it must use a privilaged port to do user
    name authentication.

Often code that was never designed to be suid, is made suid, durring porting to solve file access problems.

We need to develope a data base that will be able to look at pairs and triplets of data, specificly: program name, suid, sgid, object accessed (why prog is suid/sgid), OS flavor/version, and flav/vers geniology.

Any sugestions on how to implement such a DB?

Step 3) Confirm hypotheses. (test and exploit flaws)
===

Step 4) Make generalizations of the underlying system weaknesses, for
        which the flaw represents a specific instance.
===

Tool Box:
=========
AGREP: I suggest everyone obtain, and install agrep from:

```
    ftp cs.arizona.edu /agrep/agrep.tar.Z
```

Agrep supports "windowing" so it can look for routines, and subroutines. It also supports logical operators and is thus ideally suited to automating the search for many of the following flaws. i.e. <psudocode>

```
      agrep WINDOW {suid() NOT taintperl()} /usr/local/*.pl
or    agrep WINDOW {[suid() OR sgid()] AND [system() OR popen() OR execlp()
            OR execvp()]} /usr/local/src/*.c
```

PERMUTATION PROGRAM: Another tool worth producing is a program to generate all possible permutations of command line flags/arguments in order to uncover undocumented features, and try to produce errors.

TCOV:

CRASH: Posted to USENET (what FTP archive?) (descrip?)

PAPERS: There are several papers that discuss methods of finding flaws, and present test suites.

1) An Emphirical Study of the reliability of UNIX Utilities, by Barton P. Miller, Lars Fredriksen, and Bryan So, Comm ACM, v33 n12, pp32-44, Dec '90. Describes a test suite for testing random input strings. Results indicated that 25% of the programs hung, crashed, or misbehaved. In one case the OS crashed. An understanding of buffer and register layout on the environment in question, and the expected input is likely to produce the desired results.
2) The Mothra tools set, in Proceedings of the 22nd Hawaii International Conference on Systems and Software, pages 275-284, Kona, HI, January '89
3) Extending Mutation Testing to Find Environmental Bugs, by Eugene H. Spafford, Software Practice and Experience, 20(2):181-189, Feb '90
4) A paper by IBM was mentioned that was submitted to USENIX a few years ago. (Anyone have a citation?).

Specific Flaws to Check For:
============================
1) Look for routines that don't do boundary checking, or verify input.
   ie: the gets() family of routines, where it is possible to overwrite buffer boundaries. (sprintf()?, gets(), etc.)
   also: strcpy() which is why most src has:

```
     #define SCYPYN((a)(b)) strcpy(a, b, sizeof(a))
```

2) SUID/SGID routines written in one of the shells, instead of C or PERL.

3) SUID/SGID routines written in PERL that don't use the "taintperl" program.)

4) SUID/SGID routines that use the system(), popen(), execlp(), or

```
   execvp() calls to run something else.

5) Any program that uses relative path names inside the program.

6) The use of relative path names to specify dynamically linked libraries.
   (look in Makefile).

7) Routines that don't check error return codes from system calls. (ie:
   fork(2), suid(2), etc), setuid() rather, as in the famous rcp bug

8) Holes can often be found in code that:
  A) is ported to a new environment.
  B) receives unexpected input.
  C) interacts with other local software.
  D) accesses system files like passwd, L.sys, etc.
  E) reads input from a publicly writable file/directory.
  F) diagnostic programs which are typically not user-proofed.

9) Test code for unexpected input. Coverage, data flow, and mutation
   testing tools are available.

10) Look in man pages, and users guides for warnings against doing X, and
   try variations of X. Ditto for "bugs" section.

11) Look for seldom used, or unusual functions or commands - read backwards.
   In particular looking for undocumented flags/arguments may prove useful.
   Check flags that were in prior releases, or in other OS versions. Check
   for options that other programs might use. For instance telnet uses -h
   option to login ...
     right, as most login.c's I've seen have:
          if((getuid()) && hflag){
                 syslog()
                 exit()
                 }

12) Look for race conditions.

13) Failure of software to authenticate that it is really communicating
   with the desired software or hardware module it wants to be accessing.

14) Lack or error detection to reset protection mechanisms following an
   error.

15) Poor implementation resulting in, for example, condition codes being
   improperly tested.

16) Implicit trust: Routine B assumes routine A's parameters are correct
   because routine A is a system process.

17) System stores it's data or references user parameters in the users
   address space.

18) Inter process communication: return conditions (passwd OK, illegal
   parameter, segment error, etc) can provide a significant wedge, esp.
   when combined with (17).

19) User parameters may not be adequately checked.

20) Addresses that overlap or refer to system areas.

21) Condition code checks may be omitted.

22) Failure to anticipate unusual or extraordinary parameters.
```

23) Look for system levels where the modules involved were written by
   different programmers, or groups of programmers - holes are likely
   to be found.

24) Registers that point to the location of a parameters value instead
   of passing the value itself.

25) Any program running with system privileges. (too many progs are given
   uid 0, to facilitate access to certain tables, etc.)

26) Group or world readable temporary files, buffers, etc.

27) Lack of threshold values, and lack of logging/notification once these
   have been triggered.

28) Changing parameters of critical system areas prior to their execution
   by a concurrent process. (race conditions)

29) Inadequate boundary checking at compile time, for example, a user
   may be able to execute machine code disguised as data in a data area.
   (if text and data areas are shared)

30) Improperly handling user generated asynchronous interrupts. Users
   interrupting a process, performing an operation, and either returning
   to continue the process or begin another will frequently leave the
   system in an unprotected state. Partially written files are left open,
   improper writing of protection infraction messages, improper setting
   of protection bits, etc often occur.

31) Code that uses fopen(3) without setting the umask. (eg: at(1), etc.)
  In general, code that does not reset the real and effective uid before
  forking.

32) Trace is your friend (or truss in SVR4) for helping figure out what
  system calls a program is using.

33) Scan /usr/local fs's closely. Many admins will install software from
  the net. Often you'll find tcpdump, top, nfswatch, ... suid'd root for
  their ease of use.

34) Check suid programs to see if they are the ones originally put on the
  system. Admins will sometimes put in a passwd replacement which is less
  secure than the distributed version.

35) Look for programs that were there to install software or loadable
  kernel modules.

36) Dynamically linked programs in general. Remember LD_PRELOAD, I think
  that was the variable.

37) I/O channel programming is a prime target. Look for logical errors,
  inconsistencies, and omissions.

38) See if it's possible for a I/O channel program to modify itself, loop
  back, and then execute the newly modified code. (instruction pre-load
  may screw this up)

39) If I/O channels act as independent processors they may have unlimited
  access to memory, thus system code may be modified in memory prior to
  execution.

40) Look for bugs requiring flaws in multiple pieces of software, i.e. say
  program a can be used to change config file /etc/a now program b assumes
  the information in a to be correct and this leads to unexpected results

```
  (just look at how many programs trust /etc/utmp)

41) Any program, especially those suid/sgid, that allow shell escapes.
```