A Day at the Races

Andreas Zeller/Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

SS 2004/CVS Revision: 1.5

The Menu

- What is a Race Condition?
- Examples
- File Access
- Temporary Files
- Locking

What is a Race Condition

A *race condition* happens when a process can change the assumptions of another process about the state of its environment because of concurrency.

```
/* Lots of include directives omitted */

unsigned char* read_file(const char* filename) {
  /* We run suid root, so we have to check access before calling open(2). */
  if (access(filename, R_OK) == 0) {
    int fd = open(filename, O_RDONLY); /* Can't fail, we checked! */
    unsigned char* buf = malloc(1024);

    if (buf != 0)
      (void) read(fd, buf, 1024); /* Ignore error */
    (void) close(fd);             /* Ignore error */
    return buf;
  } else
    return 0;
}
```

In the Meantime...

Race code	Attacker code
`access("f")`	
	`unlink("f")`
	`symlink("/etc/shadow", "f")`
`open("f")`	

In principle, it does not matter whether there is one nanosecond or one hour between `access()` and `open()`; one context switch is enough.

In practice, it's easier to attack if the window is one hour instead of one nanosecond, but it's still doable:

If you try it and it doesn't work, try it again ⇒ let a computer do it for you!

Broken passwd Command

The passwd command changes a user's password in `/etc/passwd`.

File contains user name, (encrypted) password, user ID, group ID, full name (sometimes called the GCOS field), home directory and shell, separated by colons.

```
neuhaus:abcdefghijk:7006:100:Stephan Neuhaus:/home/neuhaus:/bin/bash
```

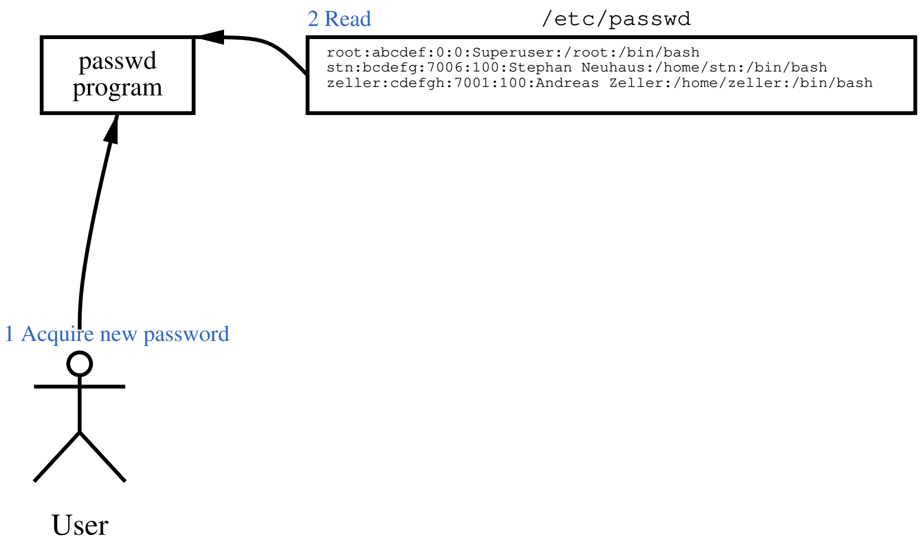
This particular passwd command allowed a user to change his password in a file that was not `/etc/passwd`, but that was given on the command line.

The passwd program must run suid root (i.e., with superuser privileges).

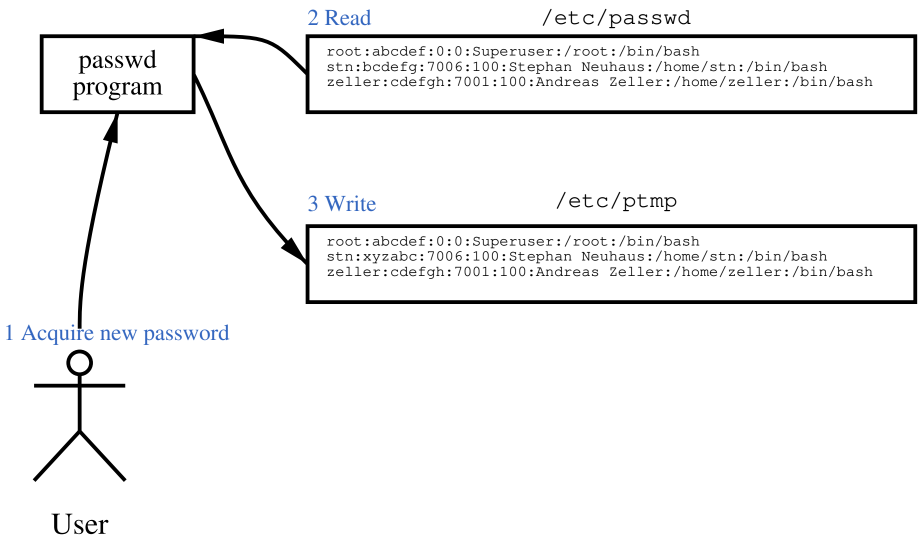
Passwd Operation

Passwd Operation

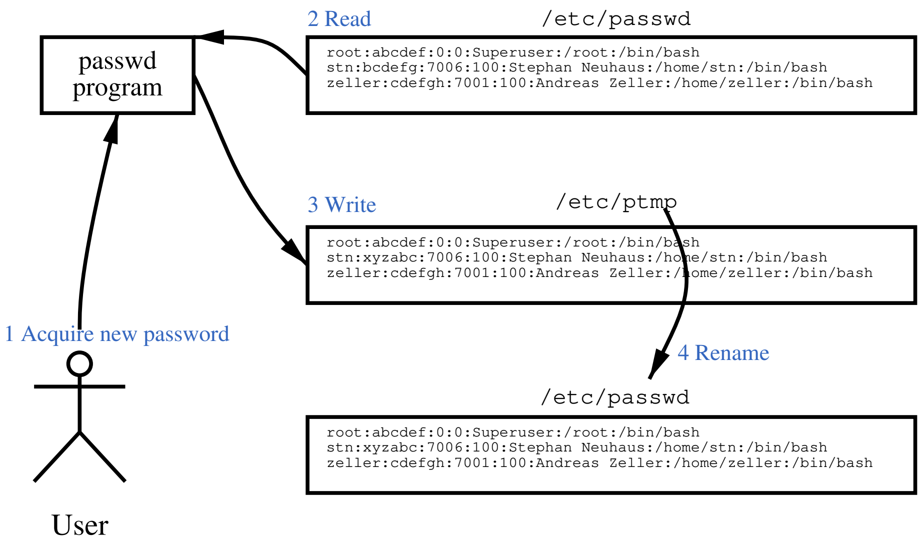
Passwd Operation



Passwd Operation



Passwd Operation



The `.rhosts` File

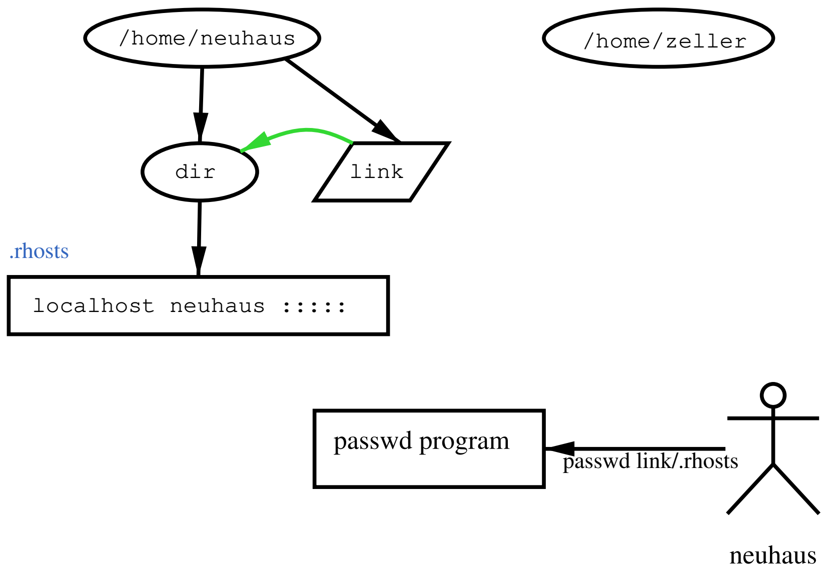
There used to be in every user's home directory a file called '`.rhosts`'.

This file contained hostname/username pairs that were permitted to use the `rsh` command to login to the user's account without a password. For example:

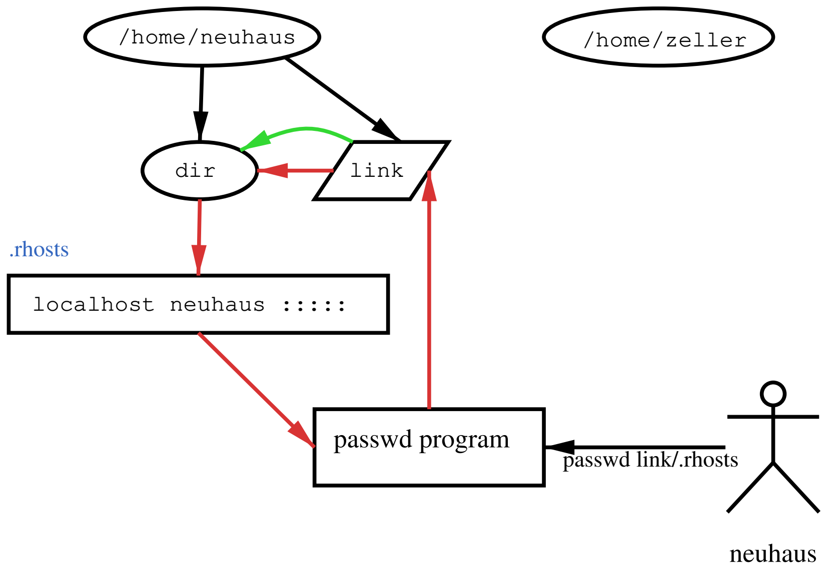
```
goscinny.cs.uni-saarland.de neuhaus
```

This would allow a user neuhaus on `goscinny.cs.uni-sb.de` to rsh to this machine and log in without a password.

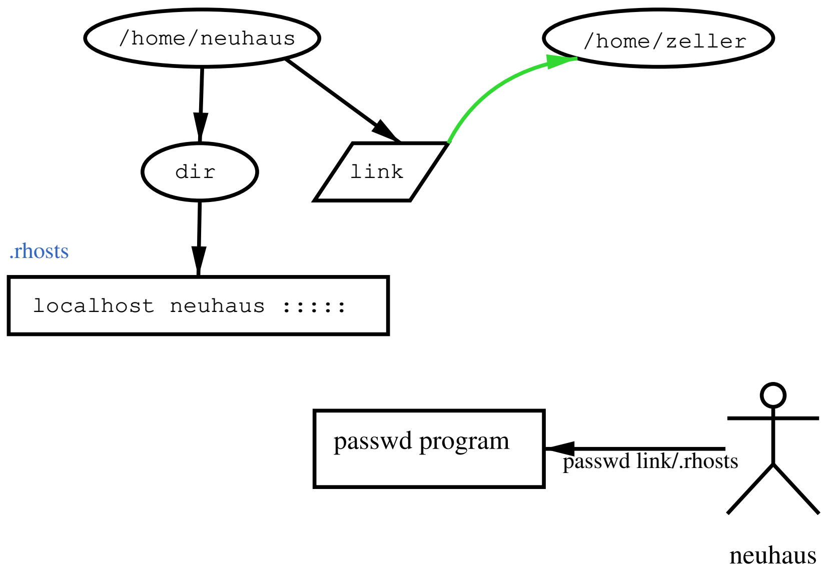
Stealing a .rhosts File



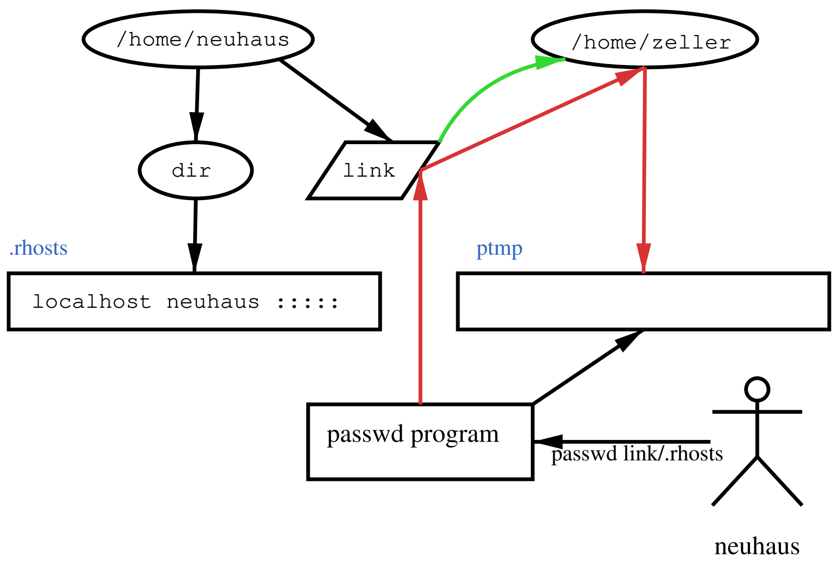
Stealing a .rhosts File



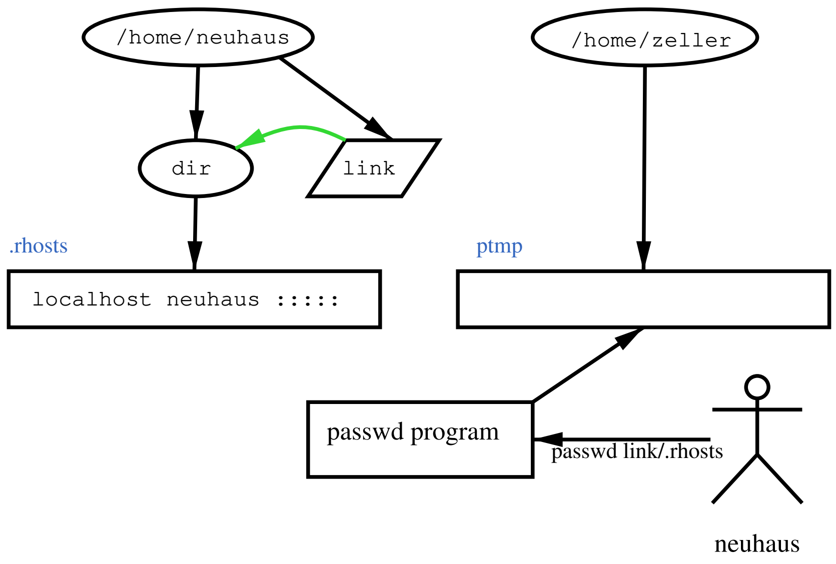
Stealing a .rhosts File



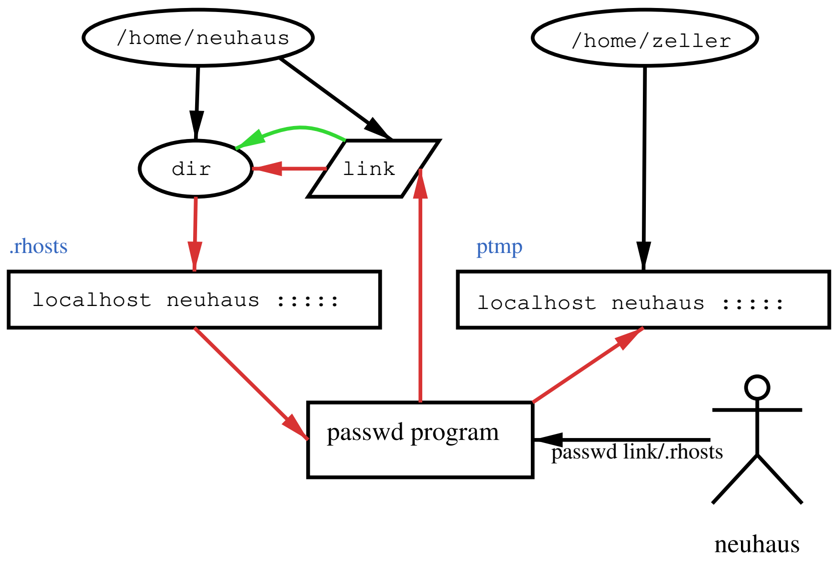
Stealing a .rhosts File



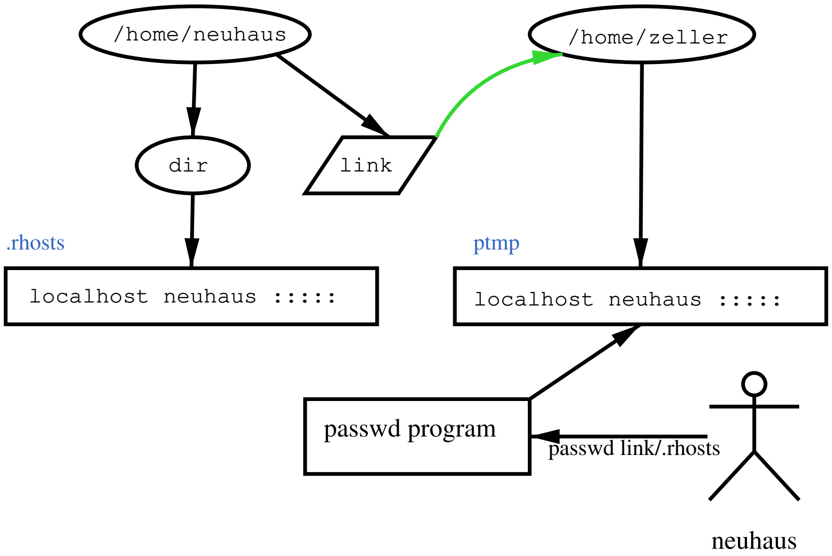
Stealing a .rhosts File



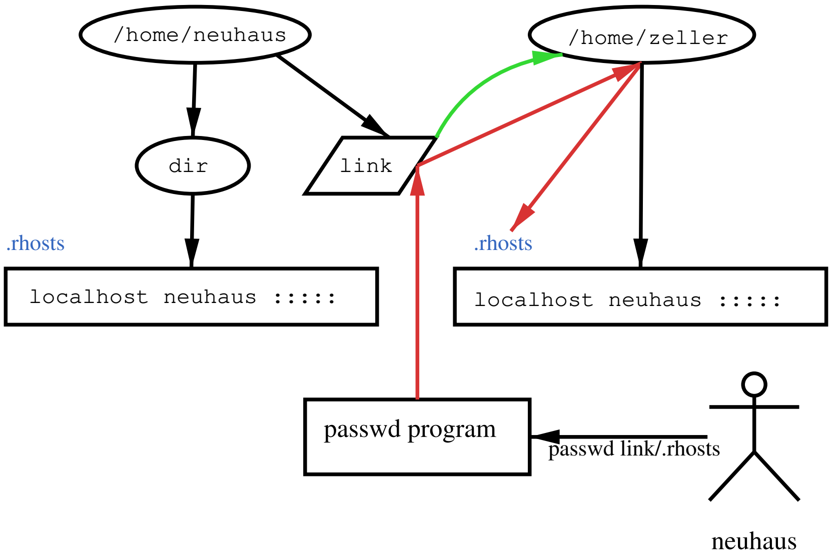
Stealing a .rhosts File



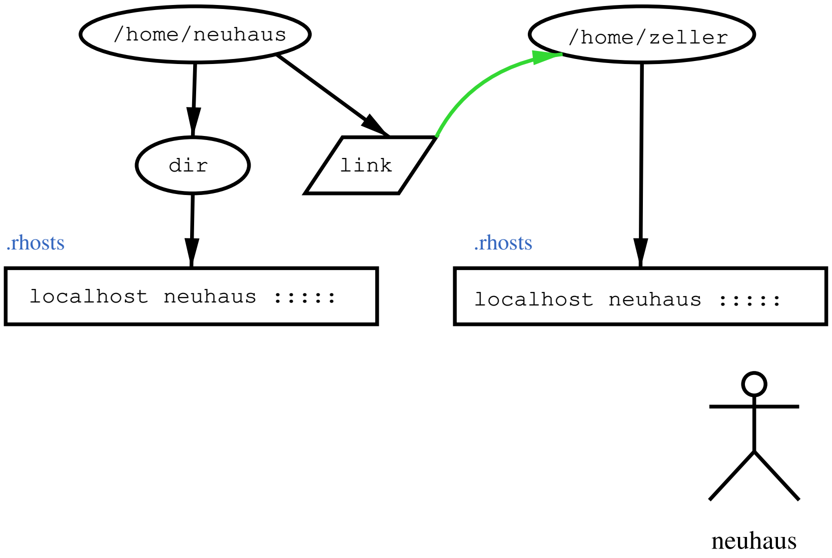
Stealing a .rhosts File



Stealing a .rhosts File



Stealing a `.rhosts` File



Avoiding TOCTOU

- Check and use must be *one atomic operation*. (Or anyway as near as dammit.)
- This is not always easy to achieve, and the secure solution is not always the most obvious one.
- When working with files, try to avoid library or system calls that take a file *name* as an argument instead of a file *handle* or file *descriptor*.
- Don't do your own access checking on files. Instead, if you use a setuid program, set your euid and egid to the appropriate values and drop all extra group privileges with *setgroups(0,0)* (see references).
- *Never* use *access*(2)!

Dropping Privileges (1)

Your process runs from a suid program and *temporarily* needs to perform some action with the original caller's permissions.

Unix distinguishes between the *real* and *effective* user IDs.

The *effective* user ID (euid) is used for all permission checking; the *real* user ID (ruid) is used for bookkeeping and is changed only rarely (e.g., during login).

When I execute the passwd program, my euid would be 0 (because passwd is a suid root program), but my ruid would remain 7006.

There is also the *saved* user ID, which was added since the ruid and euid weren't enough.

UIDs in POSIX

POSIX-compliant systems have the following semantics:

- the real (user or group) ID is the real (user or group) ID
- the saved (user or group) ID is the effective (user or group) ID at the time of process execution
- the effective (user or group) ID starts out as the saved ID, and is changed by *setuid*(2) and *setgid*(2).

setuid in POSIX

```
#include <sys/types.h>
#include <unistd.h>

extern int setuid(uid_t uid);
extern int setgid(gid_t gid);
```

- If any of euid, ruid, or saved UID are 0, set ruid and saved UID to the argument.
- Set the euid of the process to the argument

POSIX suid root programs cannot temporarily drop their privileges and re-assume them afterwards!

Usually not necessary anyway: If you find that you are changing UIDs back and forth, *beware*!

Dropping privileges is tricky (see references for a tutorial)

Opening Files

How to create a file if it doesn't already exist, and how to truncate it is it does. (There is no system call that does that for you.)

Naive solution:

1. Check if file exists. If it exists, go to step 2, otherwise go to step 3

2. Open and truncate the file, and terminate.
3. Create a new file, and terminate.

This has of course a TOCTOU problem. If between steps 1 and 2 someone creates a symlink to the password file, step 2 will open and truncate it.

Opening Files: Solution

This code was copied from Viega, McGraw, *Building Secure Software*, Addison-Wesley, 2001. Code available from
http://www.buildingsecuresoftware.com

(See handout)

O_EXCL And NFS

This code works because the O_EXCL flag guarantees that the file did not already exist when the file system processes the *open*(2) call.

Unfortunately, this does not work when the directory holding the file is on a network filesystem (in Windows, this is a SMB share) using NFS versions 1 or 2.

Worse, some NFS servers don't even tell you that the file was already there.

Therefore, make sure that the file is on a local file system.

Temporary Files: Problem

Temporary files are usually created in a shared directory (`/tmp` in most Unix systems).

Temp files are susceptibe to the same problems that regular files are.

Additionally: if the attacker can guess the name of your temporary file, he or she can prepare the attack in advance (create a symlink to a certain place).

The standard C library has *tmpfile*(3), but this is often not implemented in a secure manner.

Temporary Files: Recommendation

- Pick a prefix for your file name. For example '`my_app`'.
- Generate at least 64 bits of high-quality random data
- Encode the bits so that they are printable (BASE64).
- Concatenate '`/tmp`', the prefix, and the encoded bits.
- If you plan on using *fopen*(3) in the next step, set the process's *umask* to, e.g., 0066 (see exercises). If you use *open*(2), use a restrictive *mode* argument, e.g., 0600.
- Use *open*(2) or *fopen*(3) to create the file.
- Immediately afterwards, use *unlink*(2) to remove the file from the file system. (It won't go away until you close the file again.)
- Use *fdopen*(3) if necessary to create a file pointer.
- Use the file and close it when you're done.

File Locking

Sometimes you need exclusive access to a resource, for example to make an atomic update.

There are many ways to achieve this (critical sections, mutexes, monitors etc.), all taught in lectures on Operating Systems

Unix doesn't generally have mechanisms for mutual exclusion, especially not when distributed processes are concerned.

In order to implement locking, distributed processes must cooperate.

One technique is to use lock files.

Lock Files

1. The processes agree on a file name and a location for a file (called the lockfile).
2. When a process wants to acquire the lock, it tries to create (using O_EXCL) the lockfile. If that succeeds, the process now owns the lock and proceeds to step 3. If it fails, the process waits a bit and repeats this step. (See exercises.)

3. (At this point, the process owns the lock and hence has exclusive access to the file.) The process does its normal work.
4. The process deletes the lockfile, thereby releasing the lock.

If there are distributed processes around, chances are that for one of them is on a remote machine and that the lockfile is on a NFS-mounted file system (so O_EXCL won't work).

Excursion: Unix File System (1)

In the Unix file system, a *file* is a collection of attributes and data blocks.

The attributes contain things like the file's owner and permission bits.

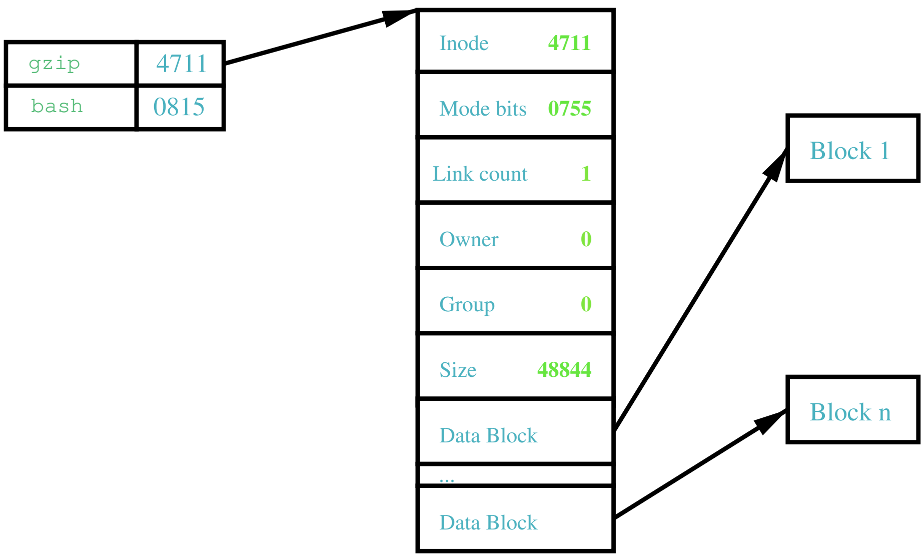
The data blocks contain the byte sequence that constitute the file. (This byte sequence has no intrinsic meaning.)

The structure that combines the attributes and data blocks is called an information node, or *inode*, which has a unique identifying number.

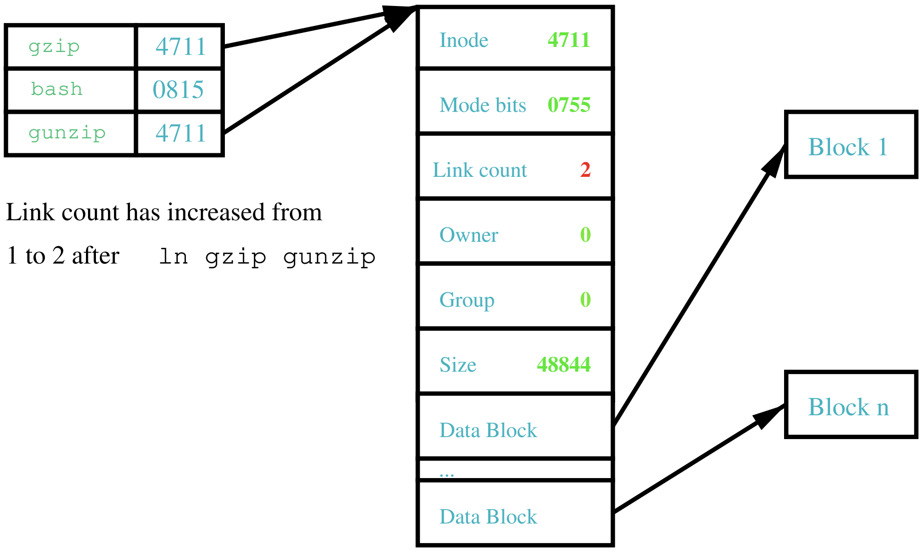
An inode contains all information about a file *except* its name.

The name is provided by a *directory*, which is a special sort of file that contains (*name*, *inode number*) pairs.

Excursion: Unix File System (2)

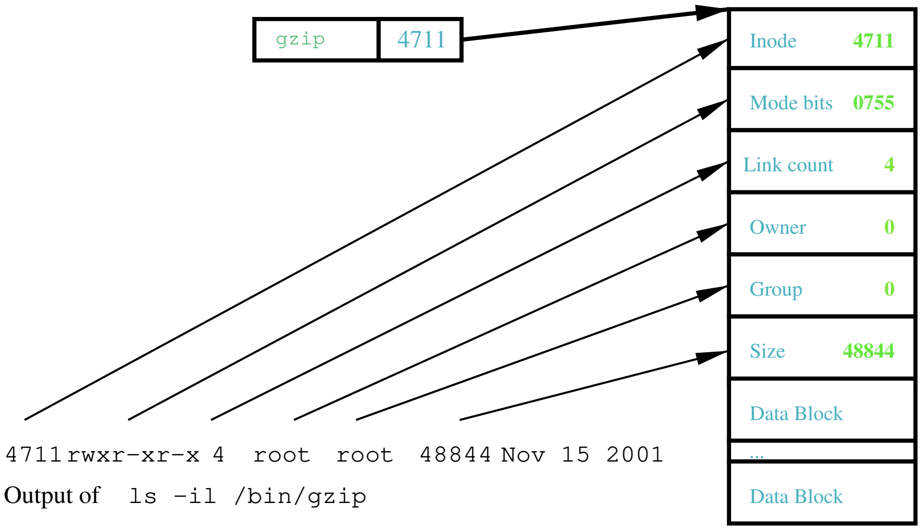


Excursion: Unix File System (2)



Link count has increased from 1 to 2 after `ln gzip gunzip`

Interpreting `ls -il`



Output of `ls -il /bin/gzip`

Lock Files and NFS: Solution (1)

1. Create a unique filename, perhaps involving the host name and process ID (it need not be unguessable)

2. In the same directory as the lockfile, create a file with the name from step 1.
3. Use *link*(2) to link the lockfile to your unique file. If the system call succeeds, the process not owns the lock and can proceed to step 5.
4. Use *stat*(2) to check the number of links on the unique file. If the link count is 2, the process also owns the lock and can proceed to step 5. Otherwise, wait a bit and return to step 2.
5. The process now owns the lock and can use *unlink*(2) on the filename created in step 1.

Lock Files and NFS: Solution (2)

The process can then proceed as usual.

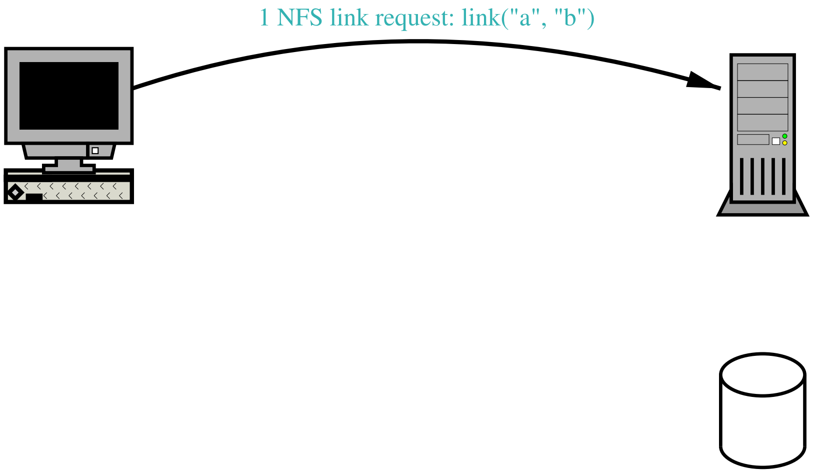
Why the funny stuff with the *stat*(2)? After all, the *link*(2) call is atomic and so either succeeds completely or fails without side effects.

Because NFS is supposed to be *stateless*, i.e., the server has no memory of outstanding client requests.

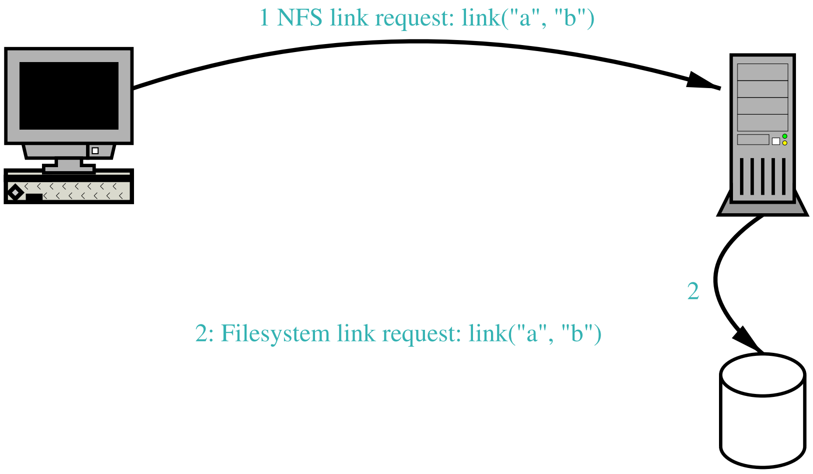
This is so that the server can continue to operate even if any component crashes and remoots in the middle of an operation.

Link System Call on NFS: Scenario 1

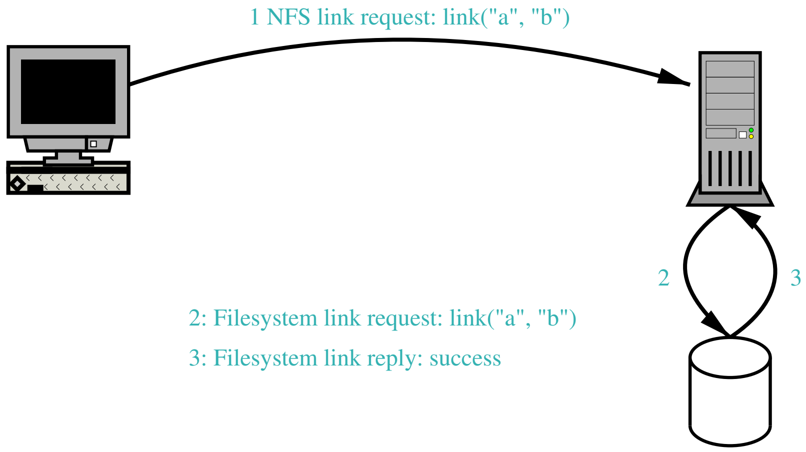
Link System Call on NFS: Scenario 1



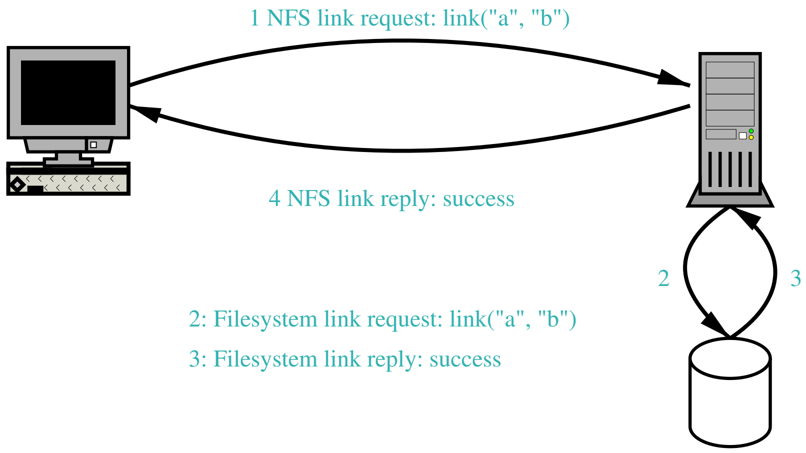
Link System Call on NFS: Scenario 1



Link System Call on NFS: Scenario 1

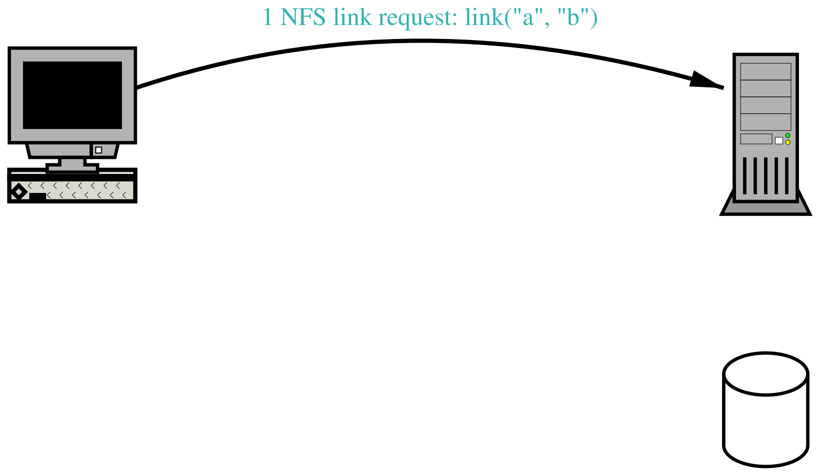


Link System Call on NFS: Scenario 1

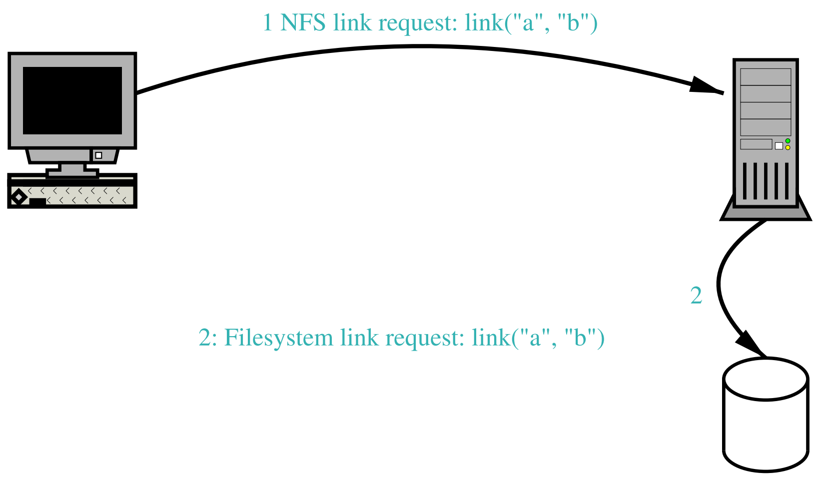


Link System Call on NFS: Scenario 2

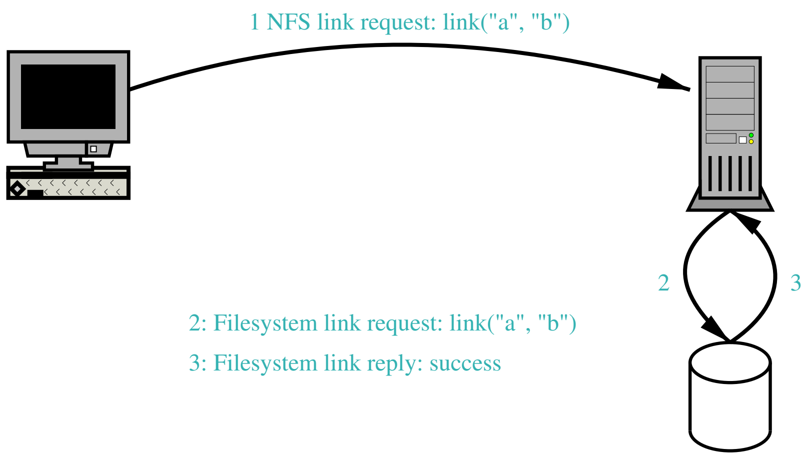
Link System Call on NFS: Scenario 2



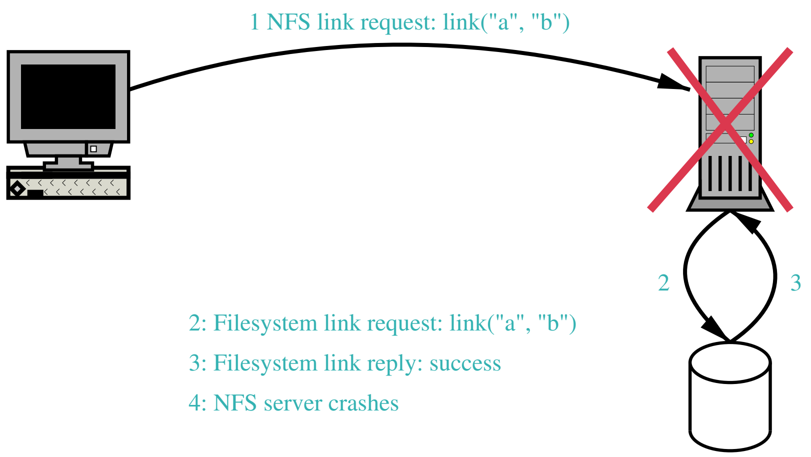
Link System Call on NFS: Scenario 2



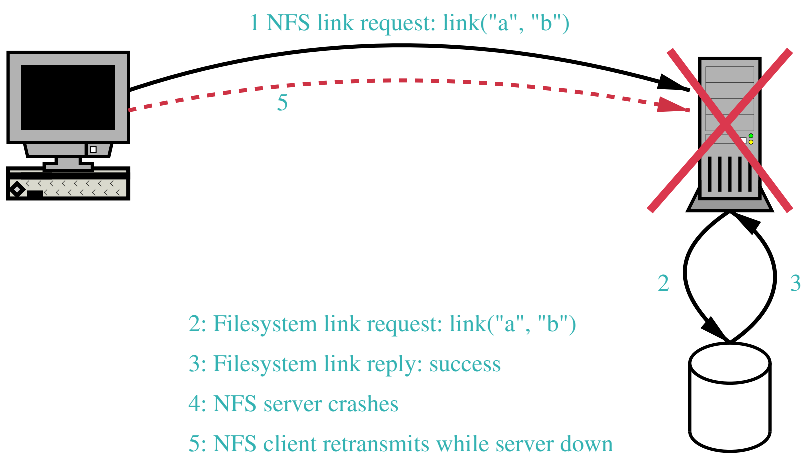
Link System Call on NFS: Scenario 2



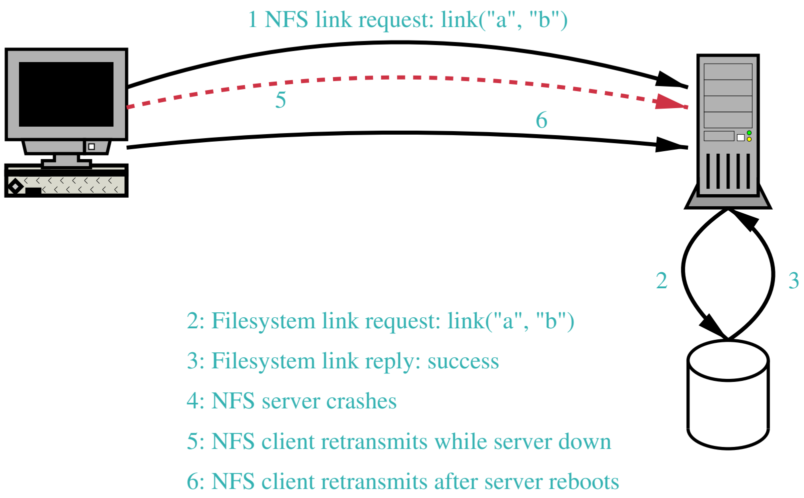
Link System Call on NFS: Scenario 2



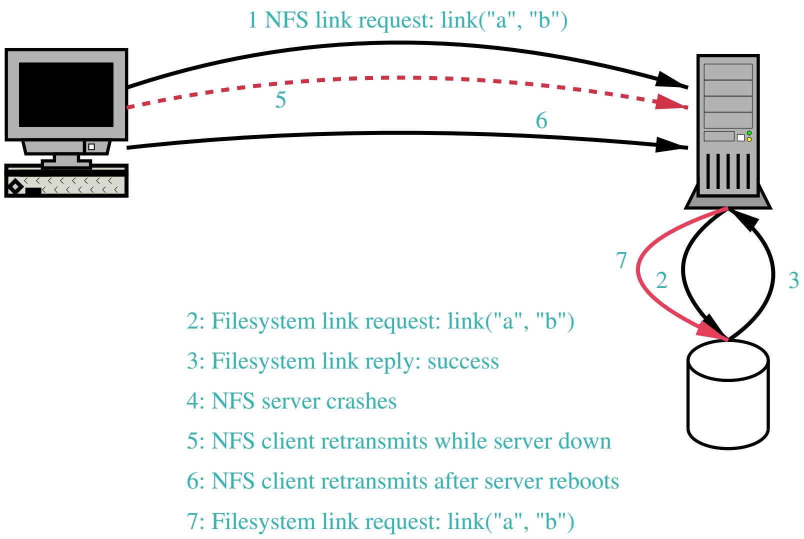
Link System Call on NFS: Scenario 2



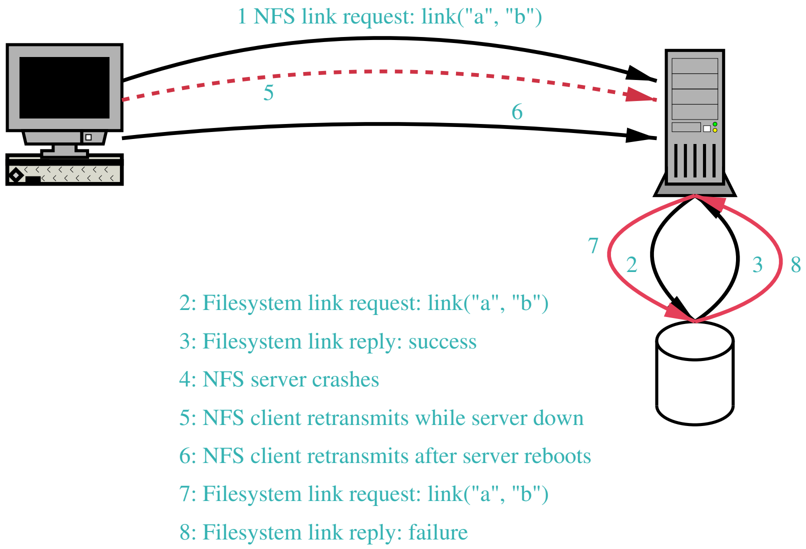
Link System Call on NFS: Scenario 2



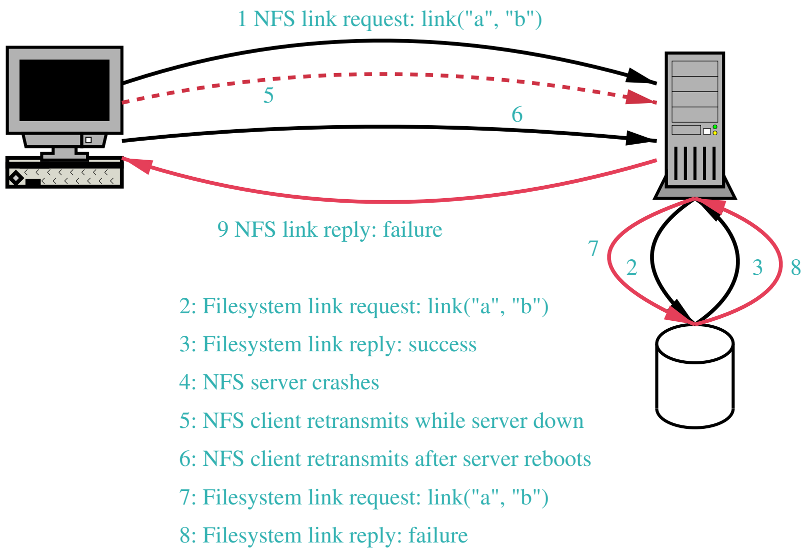
Link System Call on NFS: Scenario 2



Link System Call on NFS: Scenario 2



Link System Call on NFS: Scenario 2



Summary

- What is a Race Condition?
- Examples
- File Access
- Temporary Files
- Locking
- Obscure NFS semantics (necessary for evaluating security)
- Distributed applications extremely hard to debug (out of principle, and because of obscure application "features")

References

Matt Bishop, *How Attackers Break Programs, and How to Write More Secure Programs*, http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html

Carol Hurwitz, Scott McPeak, *Abolish Root Daemons!*, http://www.cs.berkeley.edu/~smcpeak/cs261/paper.ps, February 2001

Viega, McGraw, *Building Secure Software*, Addison-Wesley, 2001.