Lesson 3.4: Inputs

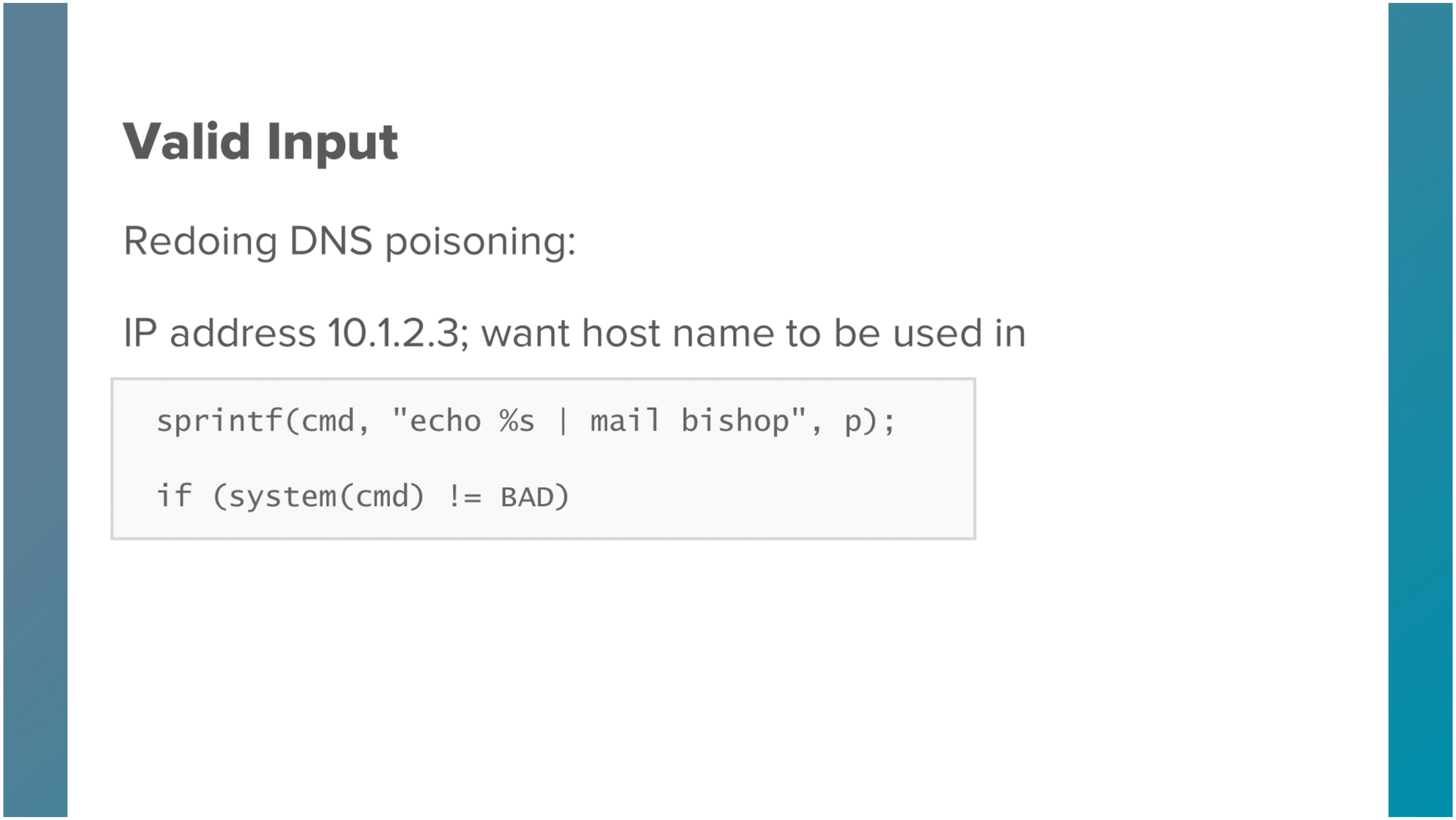
SECURITY VULNERABILITIES IN C/C++ PROGRAMMING

Inputs

Matt Bishop, Ph.D.
Professor of Computer Science,
UC Davis

UCDAVIS
Continuing and Professional Education

Slide 1: Valid Input



Valid Input

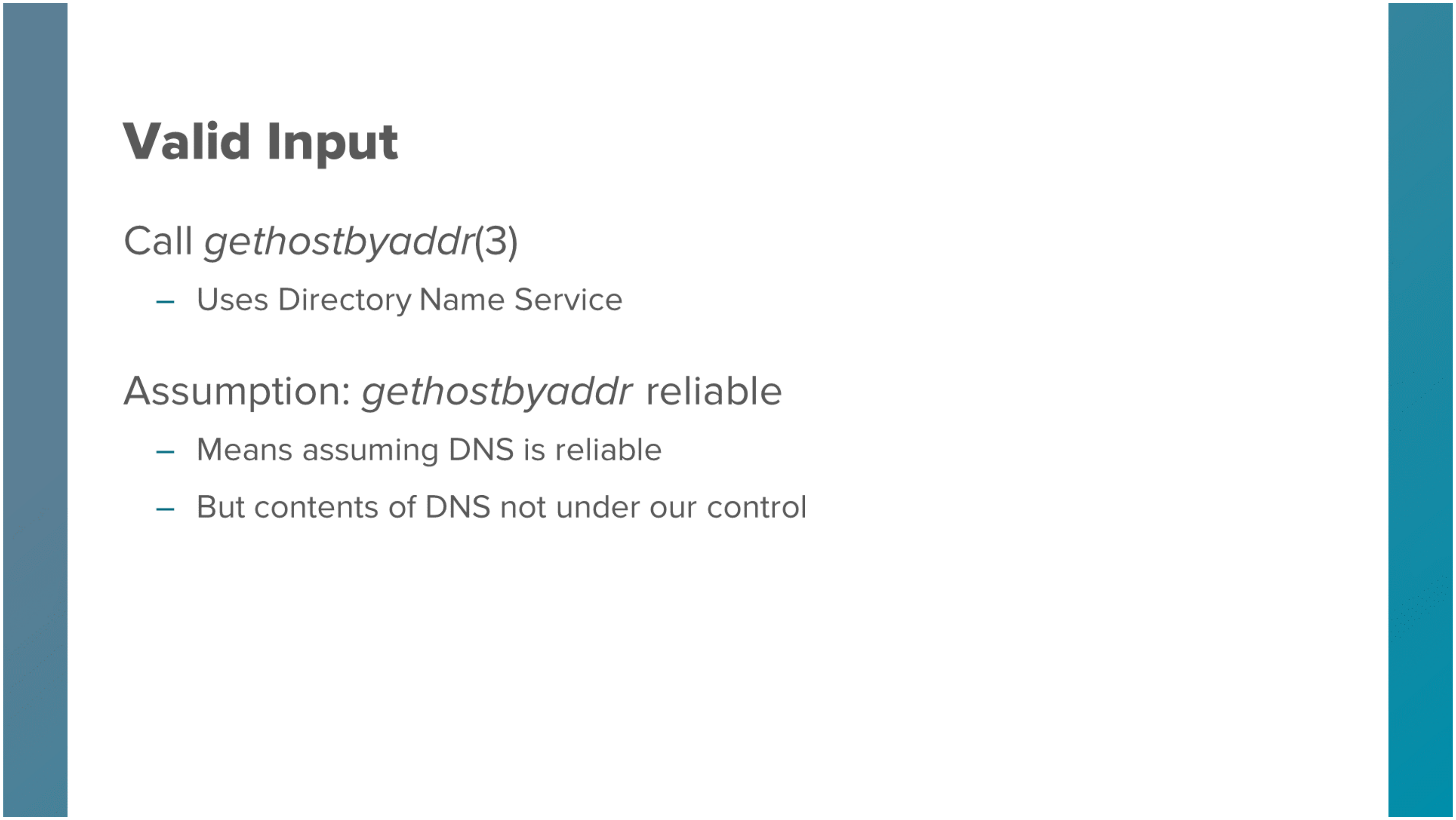
Redoing DNS poisoning:

IP address 10.1.2.3; want host name to be used in

```
sprintf(cmd, "echo %s | mail bishop", p);

if (system(cmd) != BAD)
```

Slide 2: Valid Input



Valid Input

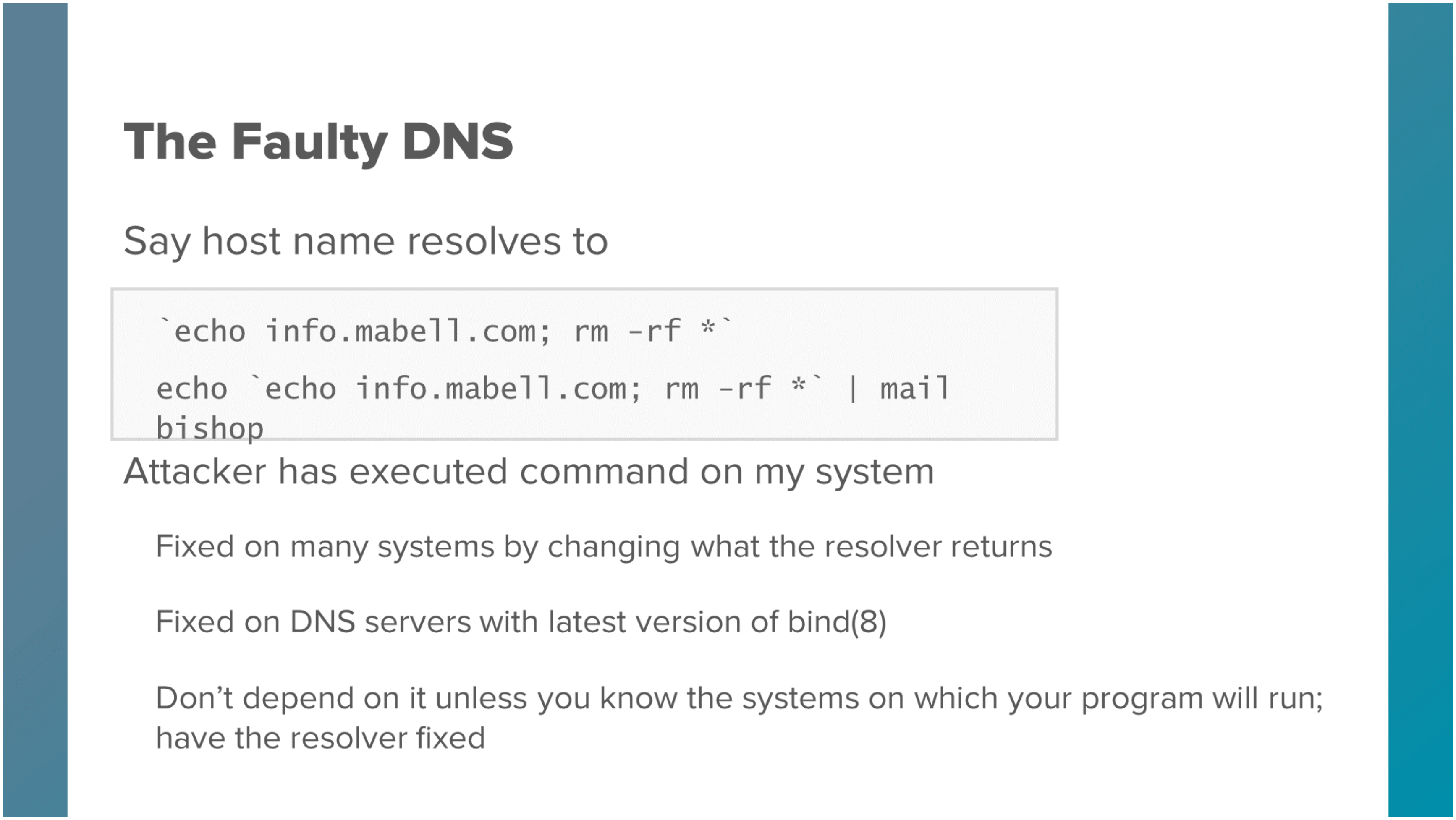
Call *gethostbyaddr*(3)

- Uses Directory Name Service

Assumption: *gethostbyaddr* reliable

- Means assuming DNS is reliable
- But contents of DNS not under our control

Slide 3: The Faulty DNS



The Faulty DNS

Say host name resolves to

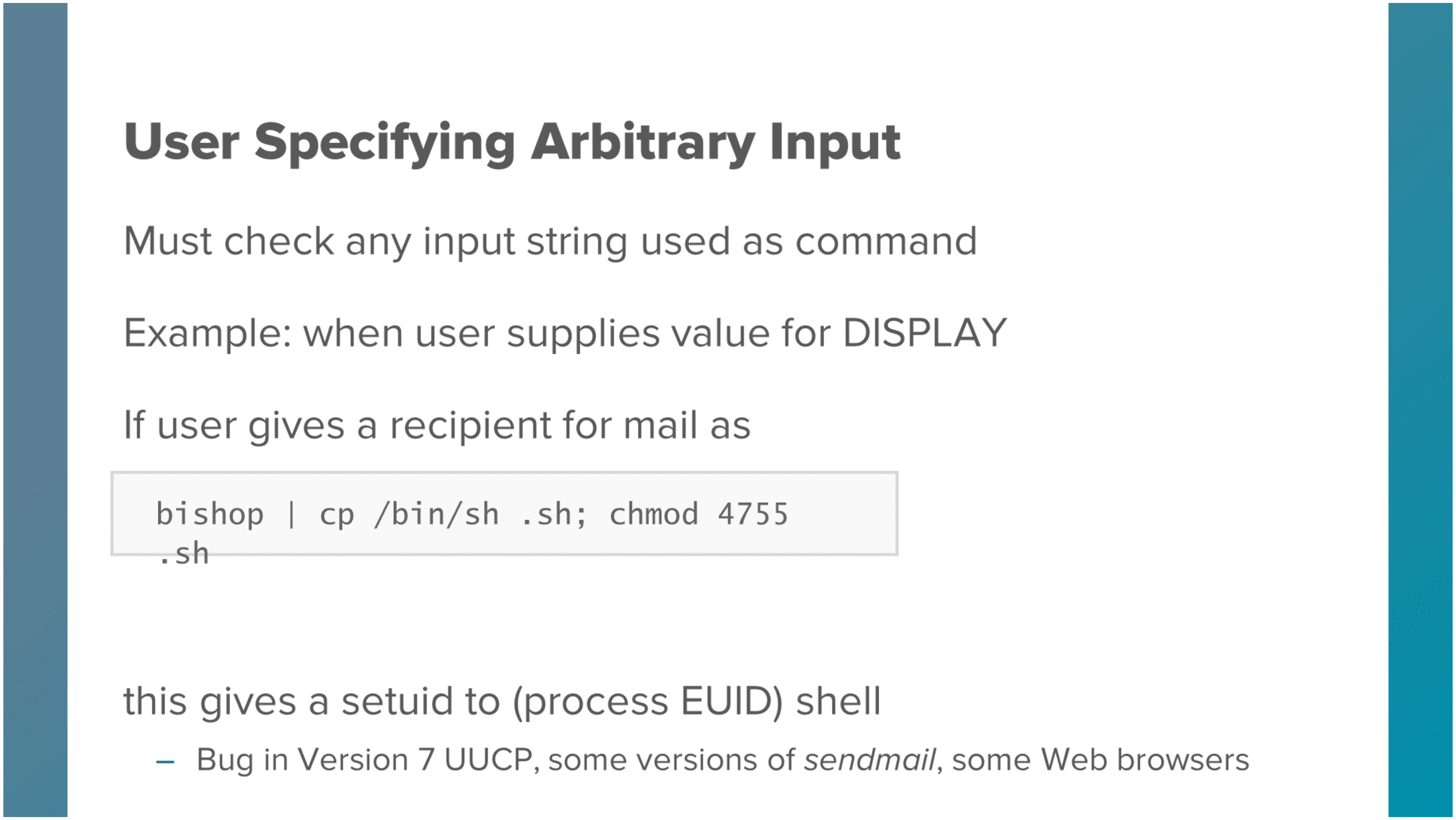
```
`echo info.mabell.com; rm -rf *`

echo `echo info.mabell.com; rm -rf *` | mail
bishop
```

Attacker has executed command on my system

- Fixed on many systems by changing what the resolver returns
- Fixed on DNS servers with latest version of bind(8)
- Don’t depend on it unless you know the systems on which your program will run; have the resolver fixed

Slide 4: User Specifying Arbitrary Input



User Specifying Arbitrary Input

Must check any input string used as command

Example: when user supplies value for DISPLAY

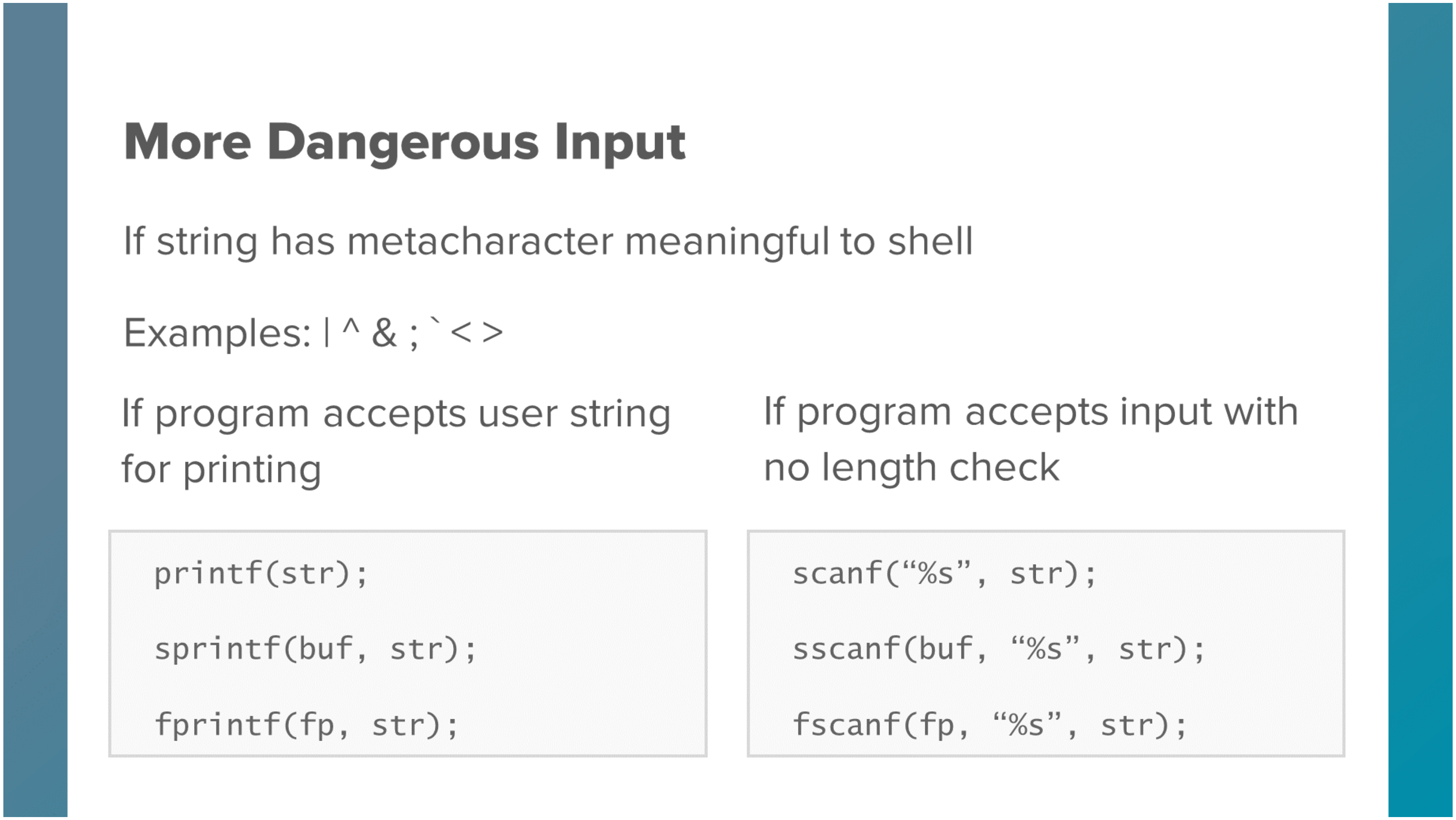
If user gives a recipient for mail as

```
bishop | cp /bin/sh .sh; chmod 4755
.sh
```

this gives a setuid to (process EUID) shell

- Bug in Version 7 UUCP, some versions of *sendmail*, some Web browsers

Slide 5: More Dangerous Input



More Dangerous Input

If string has metacharacter meaningful to shell

Examples: | ^ & ; ` < >

If program accepts user string for printing

```
printf(str);

sprintf(buf, str);

fprintf(fp, str);
```

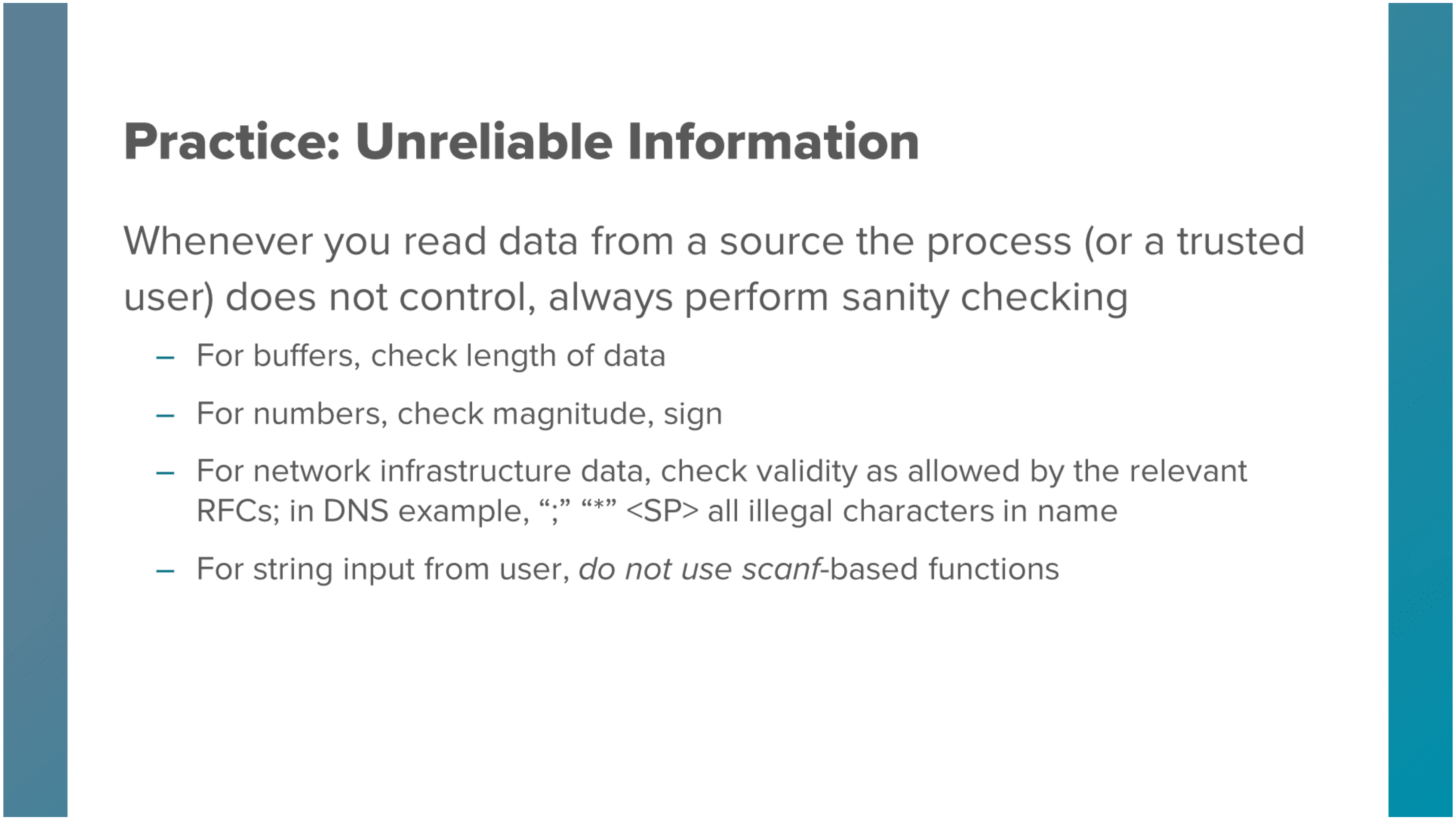
If program accepts input with no length check

```
scanf(“%s”, str);

sscanf(buf, “%s”, str);

fscanf(fp, “%s”, str);
```

Slide 6: Practice: Unreliable Information

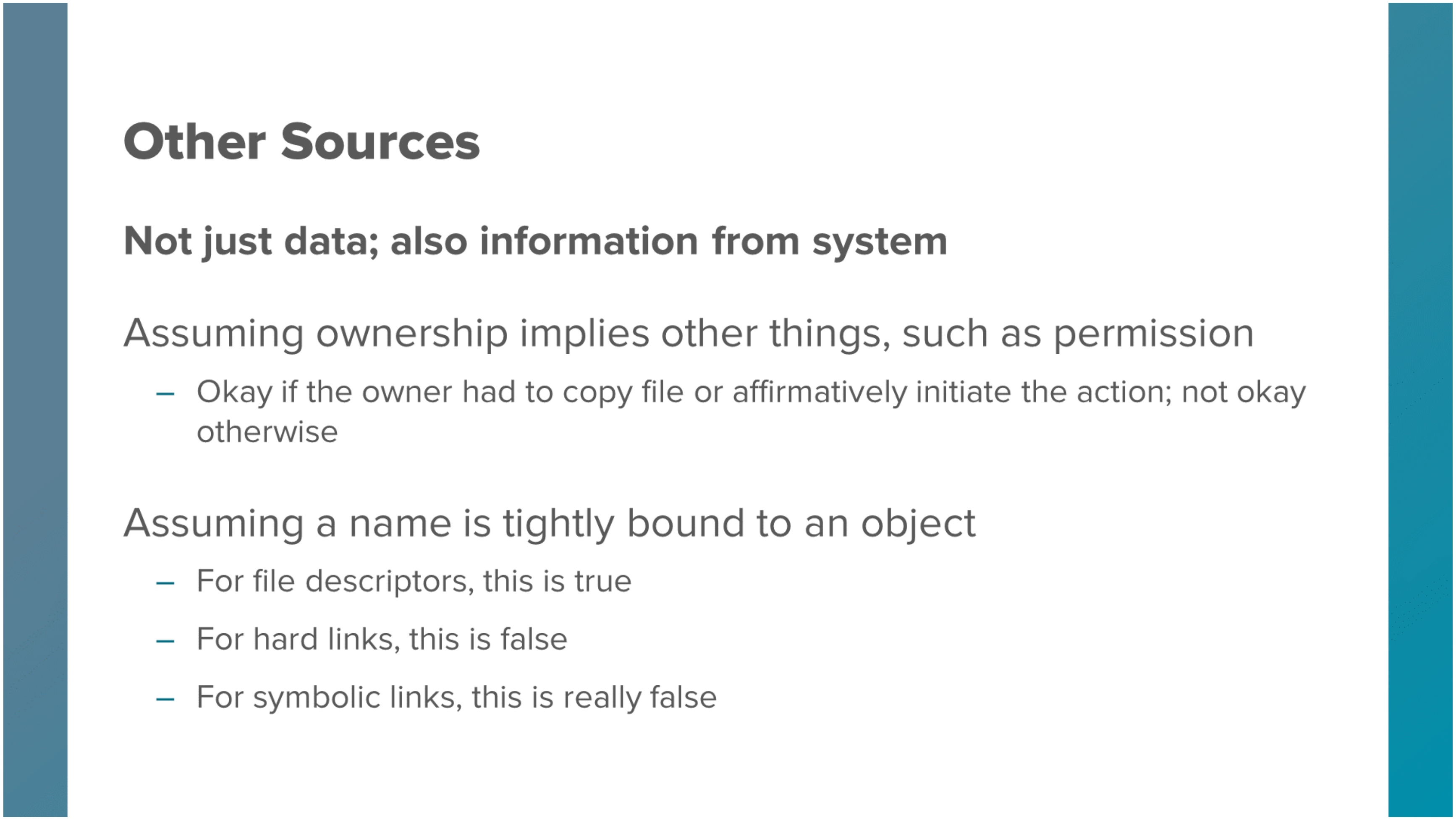


Practice: Unreliable Information

Whenever you read data from a source the process (or a trusted user) does not control, always perform sanity checking

- For buffers, check length of data
- For numbers, check magnitude, sign
- For network infrastructure data, check validity as allowed by the relevant RFCs; in DNS example, “;” “*” <SP> all illegal characters in name
- For string input from user, *do not use scanf*-based functions

Slide 7: Other Sources



Other Sources

Not just data; also information from system

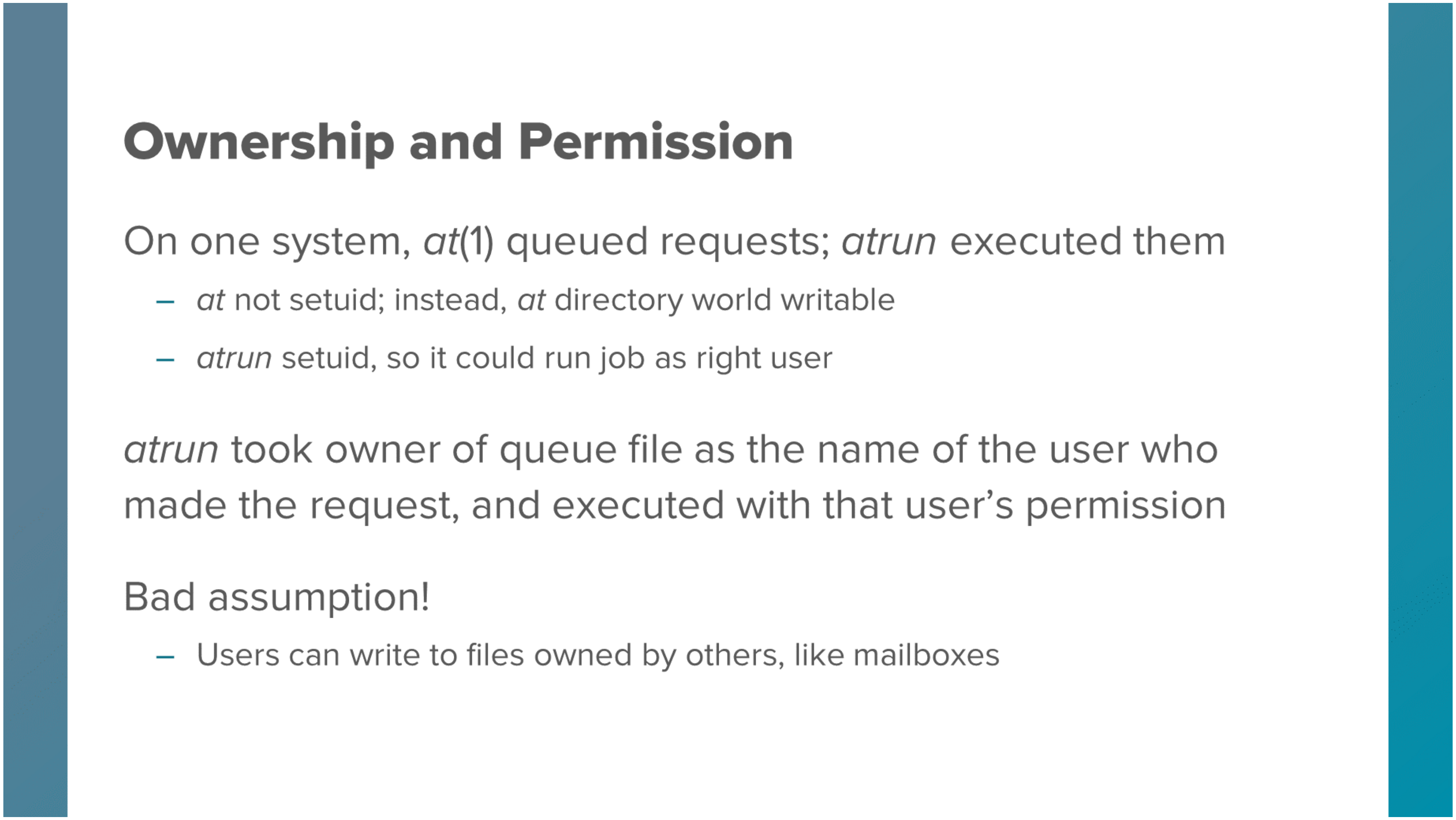
Assuming ownership implies other things, such as permission

- Okay if the owner had to copy file or affirmatively initiate the action; not okay otherwise

Assuming a name is tightly bound to an object

- For file descriptors, this is true
- For hard links, this is false
- For symbolic links, this is really false

Slide 8: Ownership and Permission



Ownership and Permission

On one system, *at*(1) queued requests; *atrun* executed them

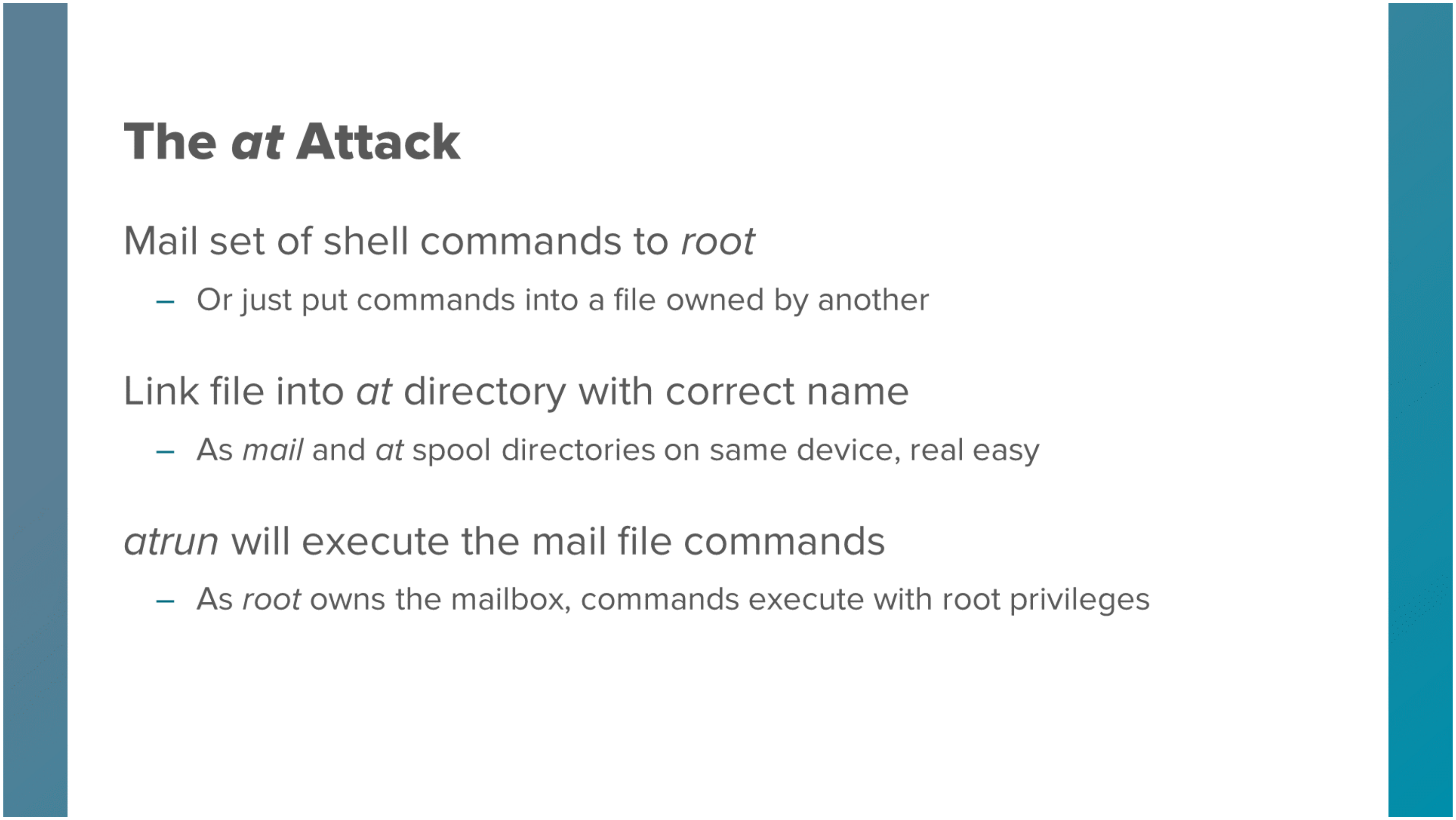
- *at* not setuid; instead, *at* directory world writable
- *atrun* setuid, so it could run job as right user

atrun took owner of queue file as the name of the user who made the request, and executed with that user's permission

Bad assumption!

- Users can write to files owned by others, like mailboxes

Slide 9: The *at* Attack



The *at* Attack

Mail set of shell commands to *root*

- Or just put commands into a file owned by another

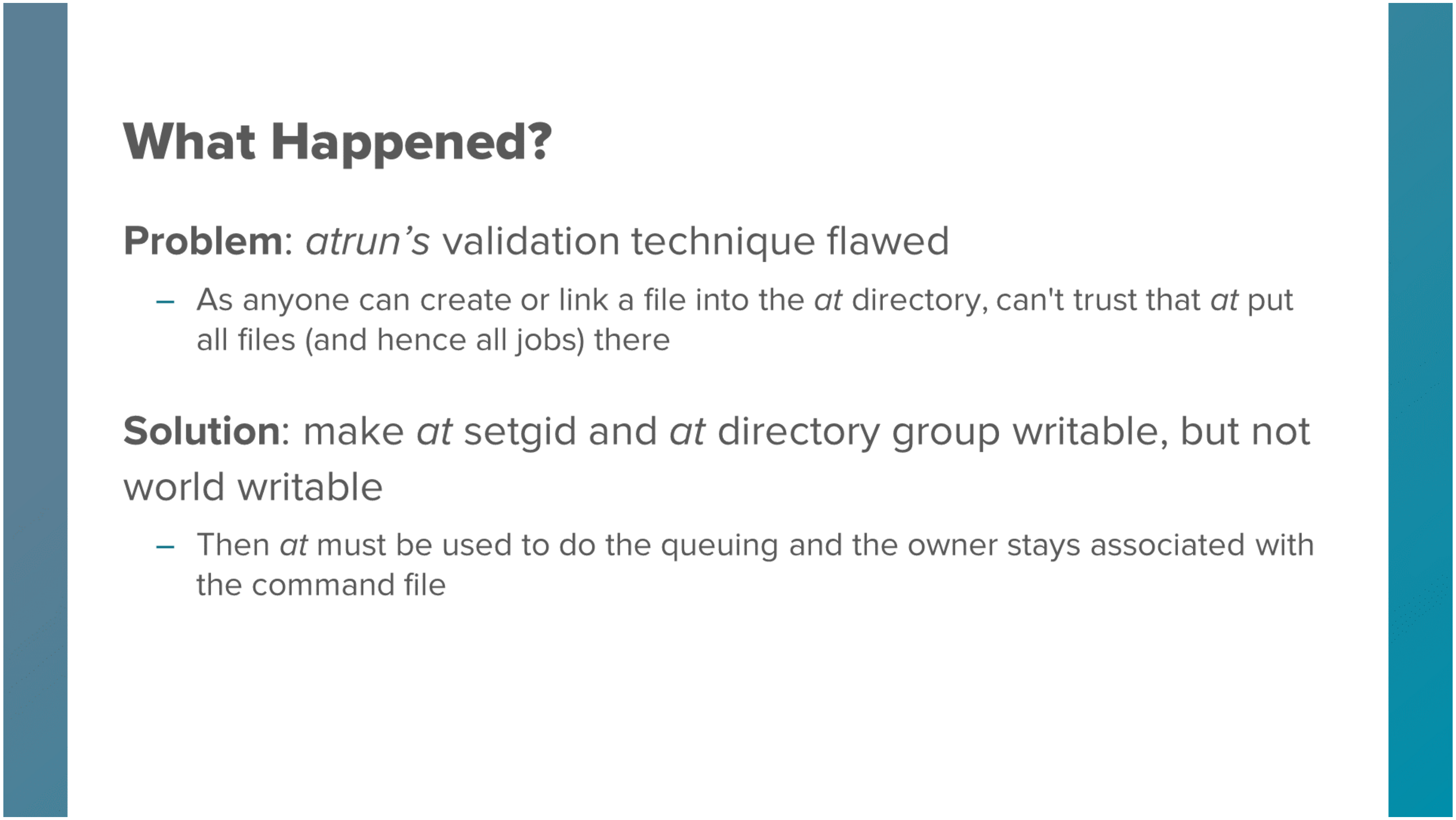
Link file into *at* directory with correct name

- As *mail* and *at* spool directories on same device, real easy

atrun will execute the mail file commands

- As *root* owns the mailbox, commands execute with root privileges

Slide 10: What Happened?



What Happened?

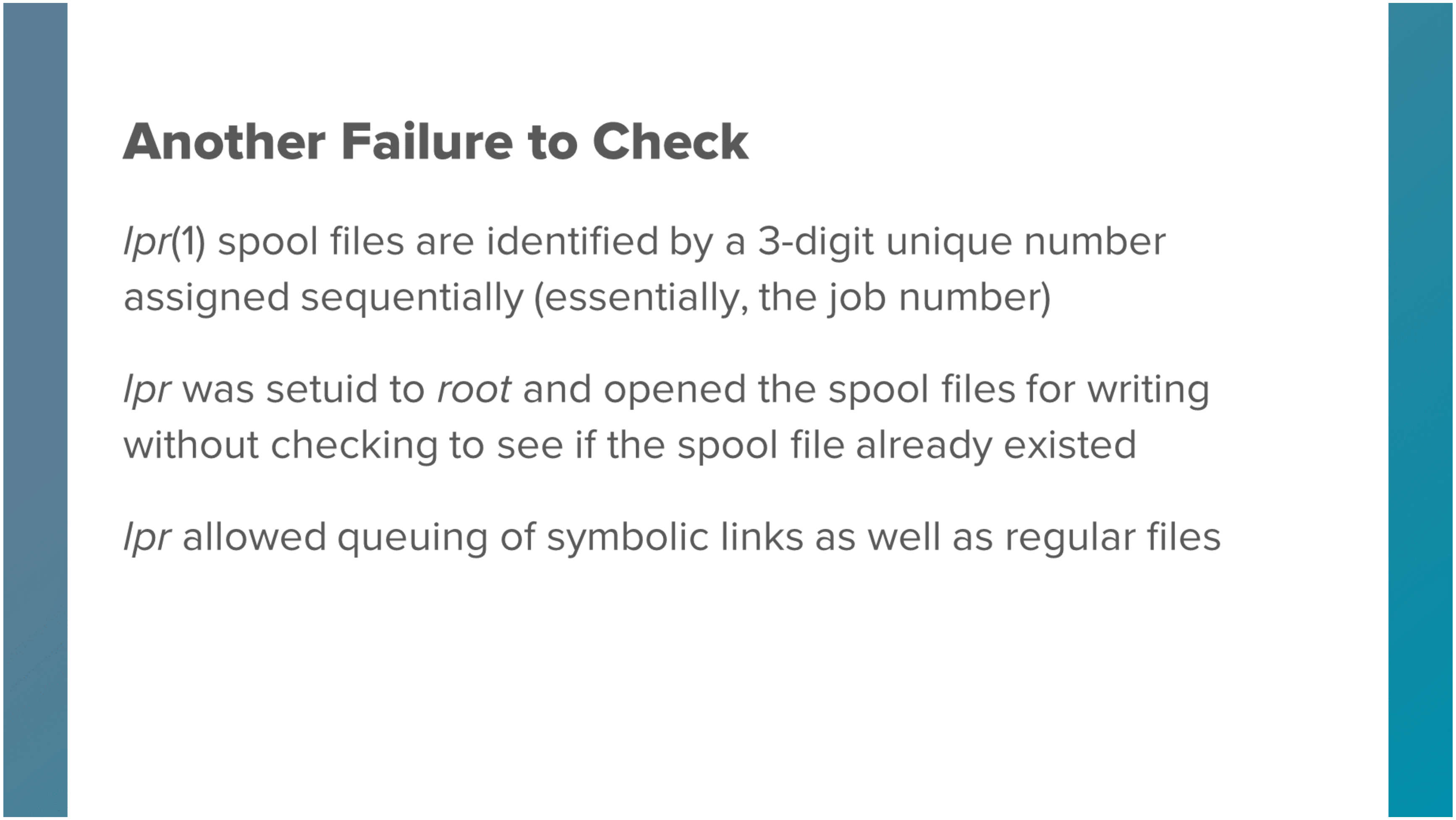
Problem: *atrun’s* validation technique flawed

- As anyone can create or link a file into the *at* directory, can't trust that *at* put all files (and hence all jobs) there

Solution: make *at* setgid and *at* directory group writable, but not world writable

- Then *at* must be used to do the queuing and the owner stays associated with the command file

Slide 11: Another Failure to Check



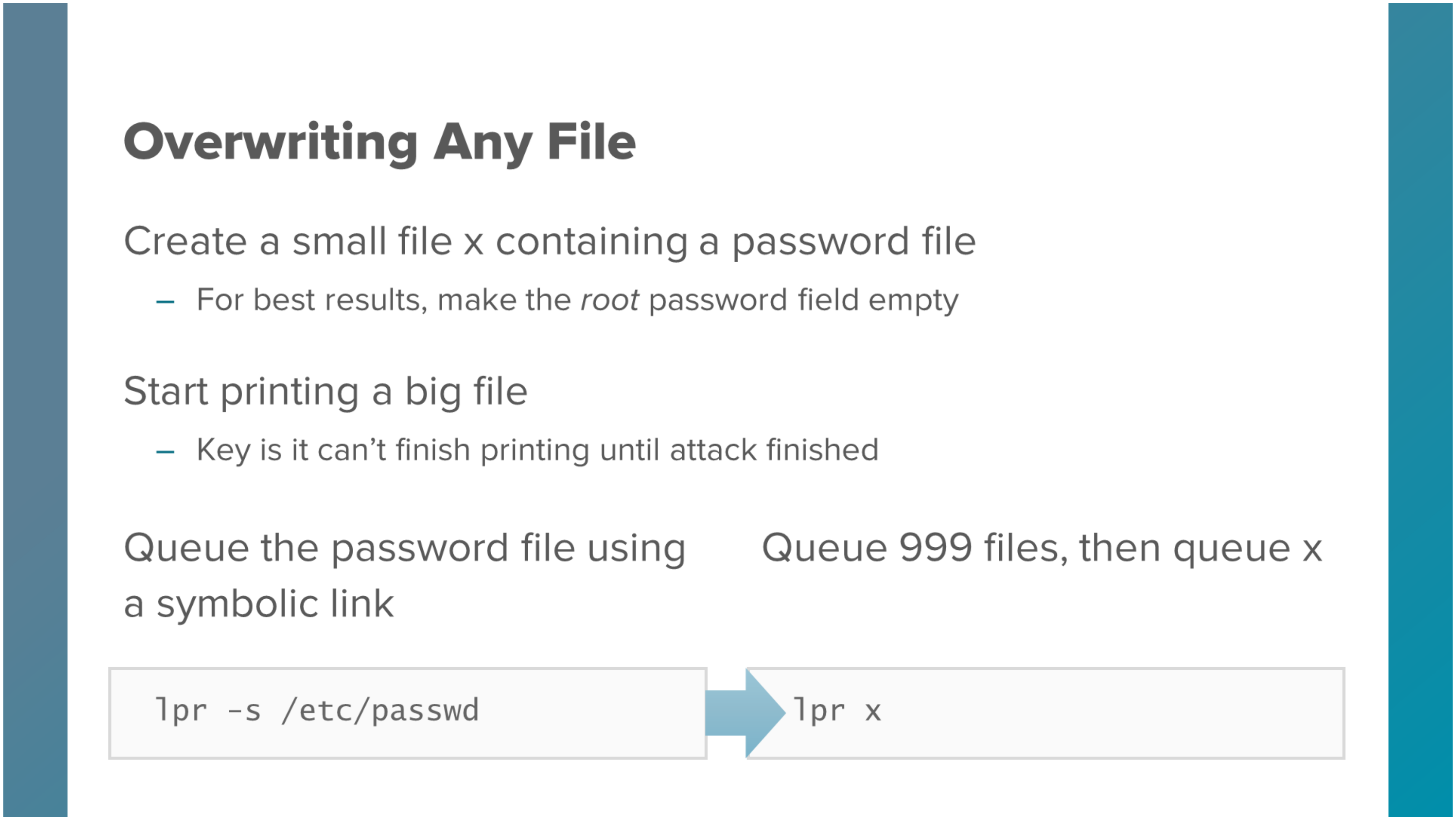
Another Failure to Check

lpr(1) spool files are identified by a 3-digit unique number assigned sequentially (essentially, the job number)

lpr was setuid to *root* and opened the spool files for writing without checking to see if the spool file already existed

lpr allowed queuing of symbolic links as well as regular files

Slide 12: Overwriting Any File



Overwriting Any File

Create a small file x containing a password file

- For best results, make the *root* password field empty

Start printing a big file

- Key is it can’t finish printing until attack finished

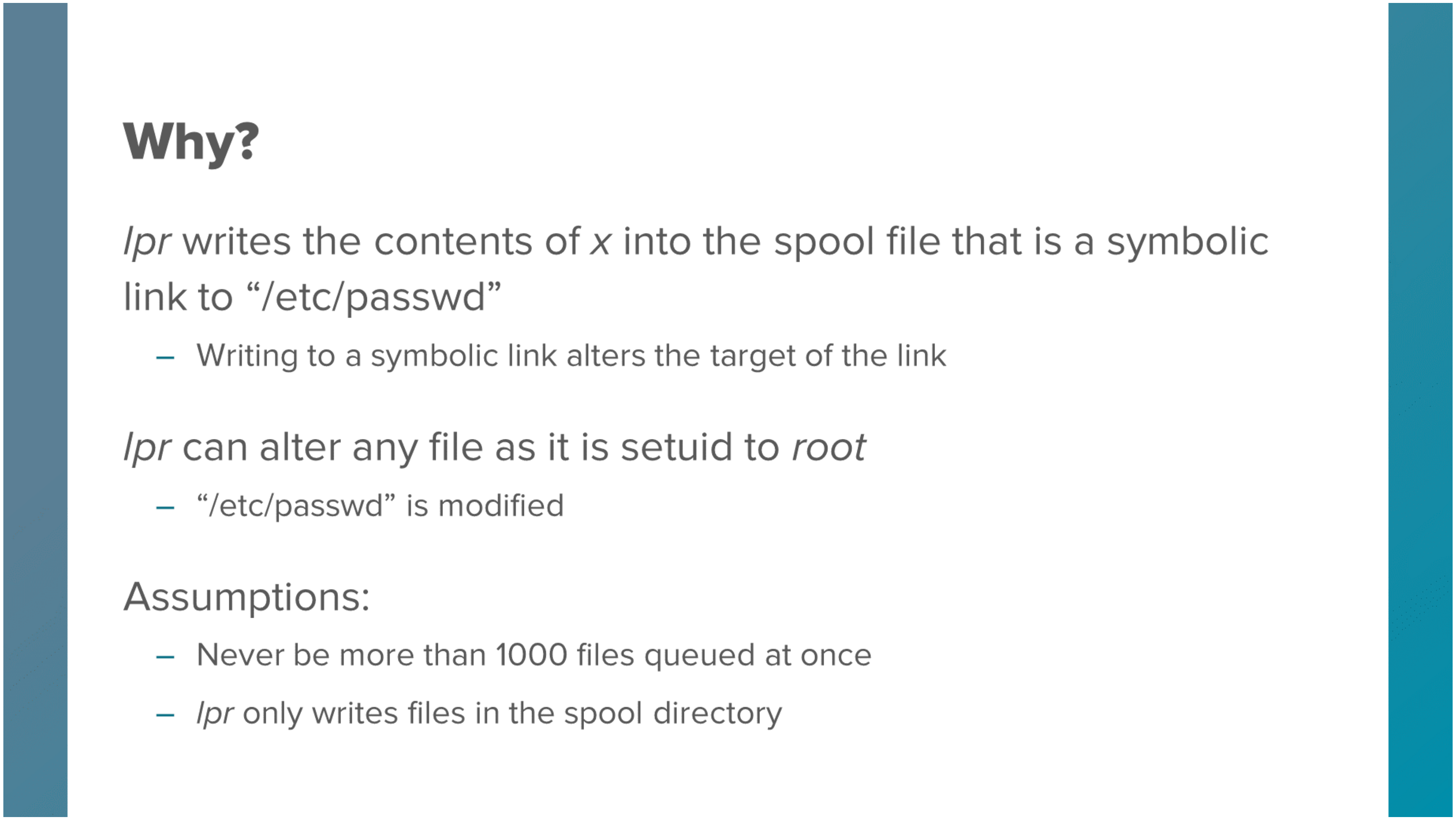
Queue the password file using a symbolic link

```
lpr -s /etc/passwd
```

Queue 999 files, then queue x

```
lpr x
```

Slide 13: Why?



Why?

lpr writes the contents of *x* into the spool file that is a symbolic link to “/etc/passwd”

- Writing to a symbolic link alters the target of the link

lpr can alter any file as it is setuid to *root*

- “/etc/passwd” is modified

Assumptions:

- Never be more than 1000 files queued at once
- *lpr* only writes files in the spool directory