Ptrace

- `pid_t waitpid (pid_t wpid, int *stat, int opt);`
  - System call also returns when debugged process stops
- ```
  int ptrace(int request, pid_t pid,
             caddr_t addr, int data);
  ```
  - Somewhat OS specific; this describes OpenBSD
  - PT_TRACE_ME – when process stopped/signaled, parent gets control via wait; also stops after execve
  - PT_READ_D, PT_WRITE_D – read/write mem in traced process
  - PT_CONTINUE – resume stopped process (addr can specify a PC address; data can specify signal)
  ```

More ptrace requests

- PT_ATTACH – start tracing a process
- PT_DETACH – continue program w/o debugging
- PT_GETREGS/PT_SETREGS – manipulate registers
- PT_GETFPREGS/PT_SETFPREGS – manipulate registers

- **ktrace – trace a process's system calls to disk**
- **systrace – trace a process's system calls and enforce policy**

Why are systems so insecure?

Sources of security holes

- **Insecure network protocols**
- **Pitfalls of C and libc (`gets`, `sprintf`, etc.)**
- **Inadequate operating systems**
  - Require many processes to be privileged,
  - Push access and sanity decisions to user level,
  - Don't provide safe ways to make such decisions.
- **Each problem worse in presence of the others**

Inadequate operating systems

- **Encourage security holes**
  1. Use all available privilege on system calls
  2. Decouple the namespace from underlying files
  3. Limited process-to-process authentication
  4. Violate the principle of least privilege
- **Careful programming is not the answer**
  - Correct code must often be convoluted
  - History shows fixes never catch up with bugs

1. System calls use all available privilege

- **Example: Wu-ftpd 2.4 (a popular ftp server)**
- **Catches SIGPIPE signal**
  - Raise privilege level to root
  - Write log file (as root)
  - Exit
- **Catches SIGURG signal**
  - Read command after out-of-band data
  - If “ABOR” longjmp out of current transfer
- **SIGPIPE + SIGURG gives root**

2. Namespace decoupled from actual files

- **Example: Root deletes old temp. files nightly:**

```
find /tmp -atime +3 -exec rm -f -- {} \;
```

- **An attack deletes any file on the system:**

	creat ("/tmp/etc/passwd")
readdir ("/tmp") = "etc"	
lstat ("/tmp/etc") = DIRECTORY	
readdir ("/tmp/etc") = "passwd"	
	rename ("/tmp/etc" → "/tmp/x")
	symlink ("/etc", "/tmp/etc")
unlink ("/tmp/etc/passwd")	

3. No process to process authentication

- **No authenticated IPC**
- **No way to grant credentials**
- **Setuid used instead of client/server model**
- **Example: Anything setuid in FreeBSD 2.1.6**
  - crt0 calls `setlocale()`
  - `PATH_LOCALE` environment variable causes buffer overrun
  - Attacker can arbitrarily corrupt stacks of setuid programs

4. Least privilege difficult to achieve

- **Even unprivileged accounts have a lot of power**
- **Many applications must run as superuser**
  - login, su, ftpd, mountd, sshd, popd, imapd, cvs, …
  - A bug in any one of these completely compromises a system
- **Simple example: old AIX and Linux login**
  - Rlogind and login both have root privilege
  - Rlogind gives login `-f` flag if user already authenticated
  - Logging in as user `-froot` gives root without password
  - Login never should have been root in the first place!

Correct code must often be convoluted

- **Example: SSH 1.2.12**
- **Reads root files and writes user files**
- **To avoid complex race conditions:**
  - Reads root-owned secret key file first
  - Drops all privileges before writing user file
- **Dropping privs allows user to "debug" SSH**
  - Secret host key could be compromised
- **The fix is painful: restructure into 3 processes!**
- **Newer SSH daemons separate privilege even more**
  - Requires re-creating one process's heap in another

This is a fundamental problem!

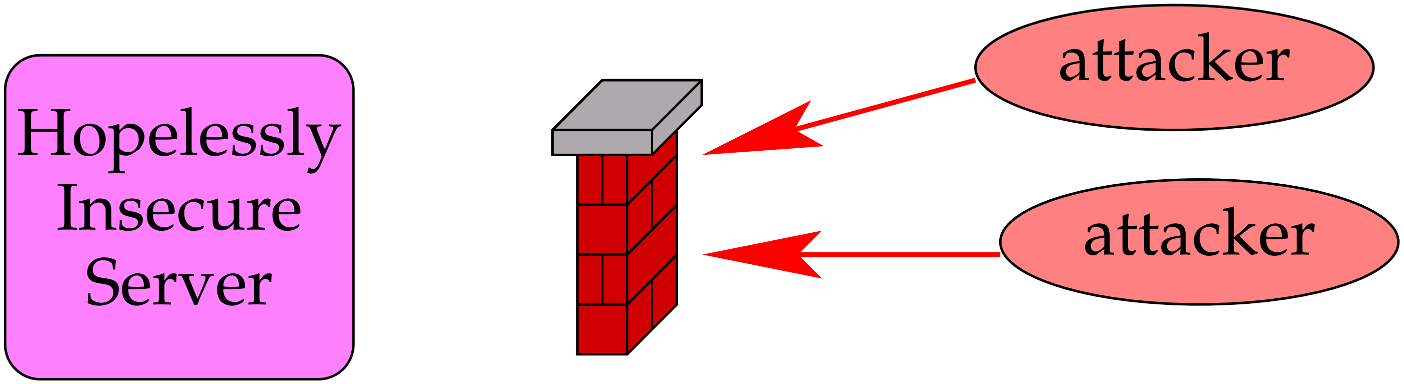
- **Can’t just blame application writers**
- **Operating systems deficiencies**
  - Require many processes to be privileged,
  - Push access & sanity decisions to user level,
  - Don’t provide safe ways to make such decisions.
- **The result**
  - Correct code must often be convoluted
  - Can’t reuse code developed for untrusted applications (where improbable case can be ignored)
  - Authentication happens in many places on one machine (login, su, sshd, popd, imapd, cvs, etc.)

On-going research at NYU, MIT, UCLA

Motivation

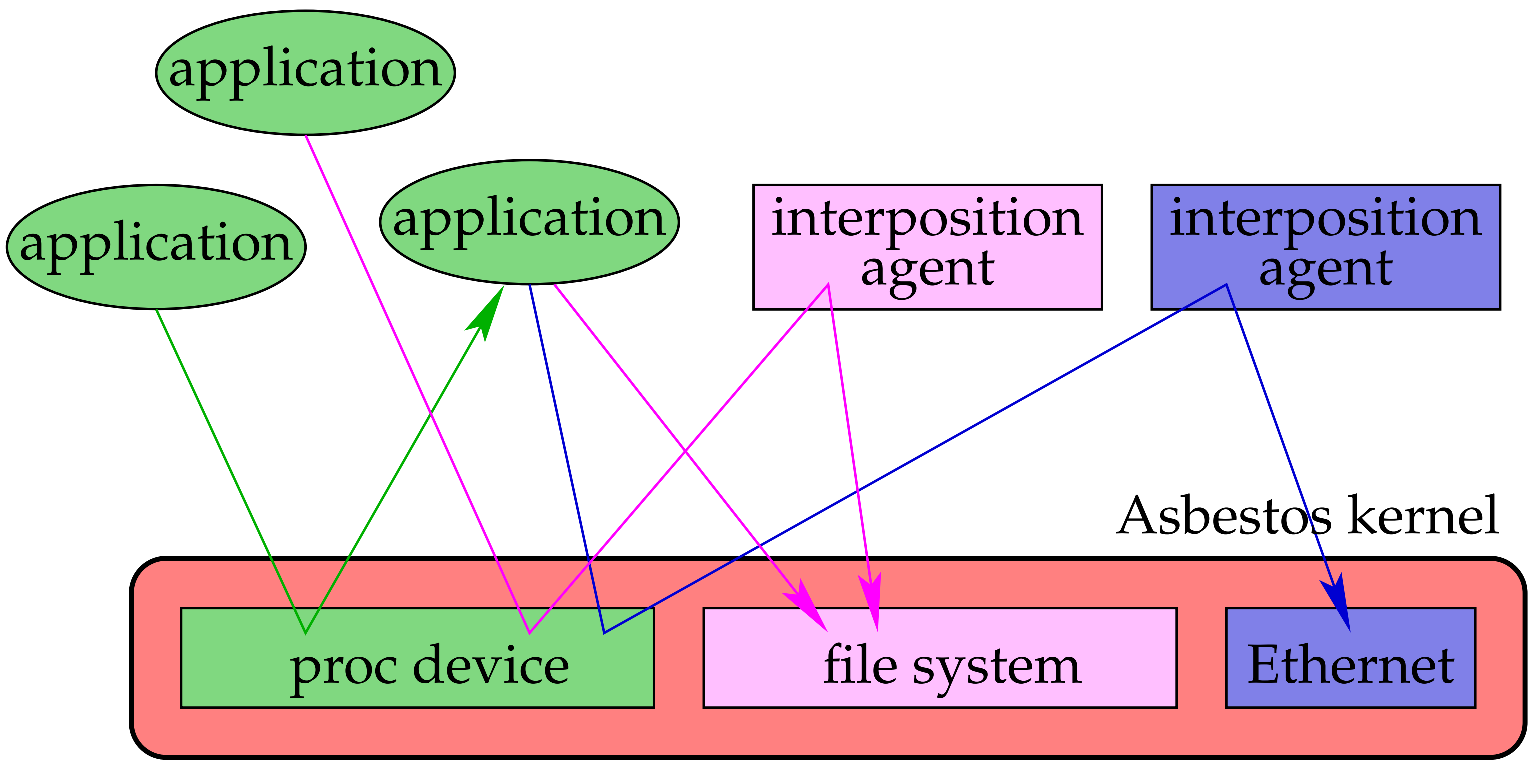
- **Most software cannot be trusted**
  - Built on error-prone OS interfaces
  - Not written by security-conscious programmers
  - Massive, complex systems no one fully understands
  - Privilege hungry—easier to implement as trusted
- **Yet this is what people develop and want to run**
- **Can such software be secured?**
  - Don't try to reason about how the application works
  - Reason about its interaction with the rest of the system

Analogy: Firewalls



- **Your machine is hopelessly insecure**
  - Can't fix software
  - *Can* reason about network traffic
- **Block interaction with network attackers**
- **Popular example of securing insecure components**
  - Of course, we know the limitations… domino effect

Asbestos



- **Push the firewall principle to individual processes**
  - Control the damage a process can do by limiting interactions
- **. . . We've just re-stated the princ. of least privilege**
  - But use simple Interposition agents to achieve it

System call interposition

- **A promising approach to controlling software**
- **Carries a performance penalty on today's OSes**
- **Q: How to understand intercepted system calls?**
  - E.g., what does `unlink ("tmp/etc/passwd")` mean?
  - Call relies on implicit state (e.g., current working directory)
- **Q: How to know what you are allowing?**
  - Meaning of call can change by the time agent executes it
- **Q: How to give agents least privilege?**
  - Agents should require all privileges
  - Combine multiple agents & not worry about order/trust?
- **Q: How to craft policies across resource types?**

Goals of an interposition-friendly OS

- **How to design syscall interface for least privilege?**
- **Unambiguous operations**
  - Effects of an intercepted operation must be clear, immutable
- **Uniform naming and interfaces for all resources**
  - Files, sockets, signals, devices, processes (think Plan9)
- **Must be able to interpose on any system request**
  - Nonbypassable, transparent
  - Object-level granularity (e.g., not servers on ports)
- **Least privilege for *interposer* & apps both**
  - Sometimes agent must make access control decisions
  - Better if agent's task is to satisfy privilege hungry application w/o privileges, through virtualization

Asbestos interface

- **Every interaction is a *message* sent to a *device***
  - Every resource is a device—even uesr processes
  - Messages like a network file system protocol
- **Messages are addressed to *handles***
  - Many-to-one mapping of handles onto devices
    (Think V object IDs or Plan9 files)
  - Each process possesses some set of handles, tracked by OS
    (Like capabilities)
- **Message format: ⟨dest, type, data, grant[], show[] ⟩**
  - grant transfers handles between procs
  - show proves possession (for credentials)

Handle possession rule: *A process must possess all the handles included in each message it sends.*

Mount device

- **Don't want to interpose on every system call**
- **May want to combine multiple interpositions**
  - Order shouldn't matter for non-overlapping goals
- **Each process has a *mount table***
  - Contains mappings *handle*$\rightarrow$ $\langle$device, *target-handle*$\rangle$
  - Any time *handle* is received in grant or show, kernel substitutes *target-handle*
  - Must possess both handles to install or remove mount entry
- **Allows surgical insertion of interposition agents**
  - Cut a process off from resources it shouldn't access
  - Emulate ones it wants but doesn't have access to

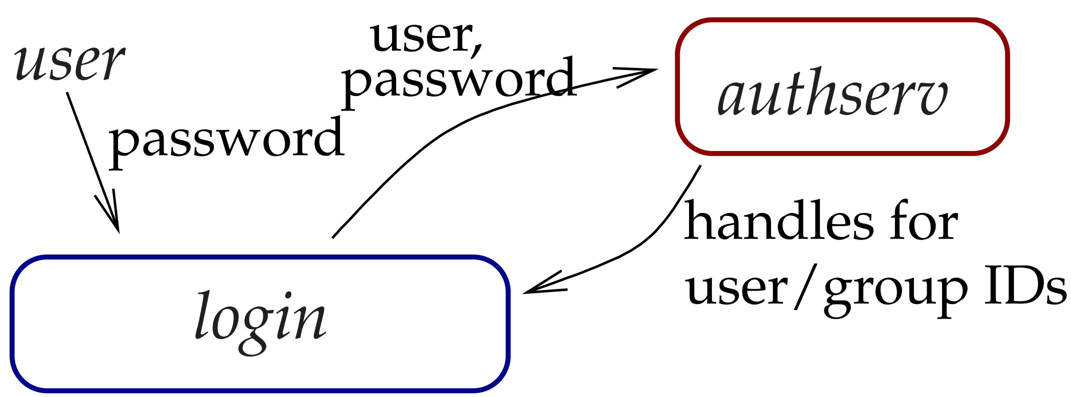
Example: Confining applications

- **Want to restricting program to certain directories**
- **Current solution: *chroot***
  - Somewhat effective (but interaction w. signals, sockets?)
  - Must be root to use it, heavy weight
  - Most applications won't work well, too privilege hungry
- **Asbestos solution: Stitch together environment**
  - Launch process with its own RAM FS for root handle
  - *mount* handles it should have access to
    E.g., `/tmp/sandbox`, `/usr/lib` for shared libs, `/proc/self`
- **Can only access functionality with handles**
  - Can't even exit w/o handle for right control node in `/proc`

Example: Blocking single-vector worms

- **Sever listening on TCP port n**
  - Often doesn't need to make outgoing connections to port n
- **Want to enforce w/o being on critical path**
  - Interposing on all socket I/O too expensive
- **Mount interposition agent on `/dev/tcp`**
  - But not in the loop for most operations
  - (Grants handles for kernel TCP device)
- **Least privilege for interposition agent**
  - Can give up its own ability to connect to port n after application listens

Example: Unprivileged login



- **Begin with no interesting handles**
- **Get username and password from user**
- **Acquire handles from authentication server**
- **Present handles in show arguments of requests**
  - Recipients can talk ask *authserv* what handles mean

Summary

- **Horrible, disgusting software is a fact of life**
- **Changing programmers is not the answer**
  - People just want to get their software working
  - Not interested in restricted programming environments, factoring applications for least privilege
- **But *can* change interfaces people program to**
  - Interposition-friendly interfaces facilitate "bolt-on" security
  - Must avoid turning people off with inconvenience
- ***Asbestos* – interposition-friendly OS interface**
  - Goal: Understand app's security w/o understanding app
  - Reason about interactions via small interposition agents
  - Challenge: Can this be done hospitably to programmers?