

CSE380 - Operating Systems

Notes for lecture 22 - 12/6/2007

© Matt Blaze

Finalized final exam information

- Final exam will be
 - Wednesday 12/19, 12:00-1400
 - Skir. Aud.
 - sorry for the late day!
- Closed book exam will cover entire course, similar format to previous exams

Operating System Security

- So far we've looked at cryptography
 - security tool, not a solution in and of itself
 - confidentiality, authentication, etc.
 - nice for protecting files and networks
- How do we secure the system?
 - and what does that mean, anyway?
- Let's look at a few parts of this:
 - authentication
 - denial of service
 - building secure software

User authentication

- You want to log in. You type a password.
 - doing this right is harder than it sounds!
- How does the system know that you typed the right password?
 - how is the password stored?
 - how is it updated on a large-scale system?
 - and how do accounts get created?

Storing passwords

- Where should the authentication server store the passwords?
- File system is the obvious choice
 - needs to be permanent
 - but the file system is insecure
- Loss of user password on one machine probably compromises others
 - people use the same password on > 1

Bob Morris & Ken Thompson: Unix Passwords

- Basic idea: store passwords using a cryptographic hash function
 - hash works in one direction - you can go from $p \rightarrow h$ but you can't easily go from $h \rightarrow p$
- Password contains the hashed version of the password
 - in fact, the file is publicly readable
 - so if an attacker gets the password file, it's no big deal

Attacks against Unix hashed passwords

- People like to use passwords they can remember easily
 - that means many of them can be guessed easily, too (e.g., dictionary words)
- Dumb attack: try to log with each guessed password
 - that will take forever, and might get noticed
- Smarter attack: do it *offline*
 - get password file, run dictionary through hash function, compare
 - you can do this on your own computer

Countermeasures

- Make users pick passwords not in the dictionary
 - even pairs of words aren't secure
- Make the hash function be as slow as possible
 - makes login a *little* slow, but dictionary attack *very* slow
- Salting: add random stuff to the password (which is included in the plaintext password) to prevent parallel dictionary attacks
- None of this stuff works very well, especially as computers are getting faster
 - plaintext passwords

Better authentication

- If user has own computer (or a small processor that can work as a *security token*) we can use that
 - no need to have a memorable/guessable secret
- Public key crypto: store secret key on user computer, public key on server
 - to authenticate, ask user to decrypt a *random challenge*
 - maybe encrypt the secret with a password for good measure
- SSH does this, for example

System administration and user authentication

- Most places that use computers have lots of them
 - more and more of them, too, now that “terminals” have been replaced by PCs and laptops
- How do you spread updated user information to all of the machines in the network?
 - this is a significant problem, since the network itself isn't secure...
- Usual solution involves authentication servers, with a secure protocol for having client machines communicate with them

Building secure software

- Many services based on the client-server model
 - anyone on the Internet can be a client
 - up to the server to authenticate, etc.
- Even if all the authentication works, the server better not have bugs in it
 - anyone on the net can send messages to it, after all...
- Problem to avoid: bug that lets anyone on the internet take control of your program, making it do things you never coded it to do

Oh, come on, that's not possible! Right?

- Some bugs that are relatively harmless on a single computer become deadly when they're on a server
- Good example is *buffer overflows*

```
char buf[128];  
int c, i=0;  
while ((c=getchar()) != EOF)  
    buf[i++] = c;
```
- Ha, I am 311t3! I own your computer d00dz

How does a buffer overflow work?

- Attacker sends input that overflows an input buffer
 - this is sent over a network, anyone can do it
- Attacker's data goes past the end of a fixed-size buffer and on to the stack.
- The stuff from attacker is executed as code...
 - game over
- Some details, but very hard to defend against
 - stack vs. heap vs. dynamic memory all have attacks when there's an overflow

Countermeasures

- Better programmers
 - don't have bugs
- Better languages
 - make it hard to overrun buffers
- Better OS
 - sandboxing server code
- Better architecture
 - prevent code on stack from being executed

Denial of Service

- Server can be dangerous even when there are no bugs
 - anyone can throw input to the server
- If input is expensive to process, server is subject to denial of service attack
- Especially bad when asymmetric
 - e.g., public key authentication
 - solving one problem makes another worse!
- Might attack bandwidth, CPU, memory or other resources

Summing up...

- Processes
- Inter-process communication
- Resource management
- Locking and mutual exclusion
- Deadlocks
- Standard problems
- Memory management
- Virtual memory
- Pages, TLBs, Caching
- I/O and Devices
- Disks
- Files and file systems
- Distributed computing and RPC
- Security and Cryptography

Operating System Design

- What does an *Operating System* do?
- How do you actually build an operating system?
- Why does everyone who tries to build an operating system end up doing such a terrible job?
- How can you tell them apart and evaluate them?

What does an operating system do?

- This was the first question we addressed in this course
- It's not so simple...
- Two schools of thought
 - Minimalist
 - Allow processes to run, manage resources
 - get out of the way
 - Maximalist
 - provide an “environment” for programs
 - window system, GUIs, libraries, tools

Mechanism vs. Policy

- What does this mean?
- Minimalist model: figure out what the common things an OS might want to do are and provide the ability to implement them
 - flexible but you need other tools
- Maximalist model: figure out the best way to do things and do it that way
 - inflexible but complete

Building an Operating System

- You've all built parts of an OS
 - and 381 students have built almost an entire OS
- A serious commercial grade OS is a major software engineering project
 - Original Unix: about 20,000 lines of C
 - kernel was about 9100 lines
 - Recent OSs: over 1,000,000 lines, including libraries and required services

Software Engineering an OS

- Original Unix was small enough that one person could understand the entire system
 - except for one comment on a particularly obscure line of code: “you are not expected to understand this”
- Modern operating systems are too large, and do too many different things, for one person to understand.
 - how can you manage such a project

Brooks: Mythical Man-Month

- If you've got a big project you need more people to help, right?
- Brooks' Law: adding programmers to a late project makes it later.
- Why is this?

Managing a large project

- Interfaces
- Testing
- Critical paths
- Debugging
- Personnel management
- Documentation
- Portability
- Coding standards
- what else?