Safe sharing

- Protecting a single computer with one user is easy
  - Prevent everybody else from having access
  - Encrypt all data with a key only one person knows
- Sharing resources safely is hard
  - Preventing some people from reading private data (e.g., grades)
  - Prevent some people from using too many resources (e.g., disk space)
  - Prevent some people from interfering with other programs (e.g., inserting key strokes / modifying displays)

Why is security hard?

- Security slows things down
- Security gets in the way
- Security adds no value if there are no attacks
- Only the government used to pay for security
  - the Internet made us all potential victims
- Bugs R Us

Principle of Least Privilege

- Figure out exactly which capabilities a program needs to run, and grant it only those
  - not always easy, but: start out granting none, run program, and see where it breaks. add new privileges as needed.
- Unix: concept of root is not a good example of this
  - some programs need to run as root just to get a small privilege, such as running with a port < 1024
    - e.g., ftpd

Principle of Complete Mediation

- Check every access to every object
  - in rare cases, can get away with less (caching)
    - but only if sure nothing relevant in environment has changed...and there is a lot that's relevant!
- e.g., NFS and file handles
  - NFS is not a good example of complete mediation
    - NFS protocol:
      - client contacts remote "mountd" to get a filehandle to a remotely exported NFS filesystem
        - this is done when remote system is mounted
        - remote mountd checks access control at mount time
        - filehandle is a capability: client presents it to read/erase file
          - access control is not checked after mount time!
        - use network sniffer to get filehandle
          - access exported filesystem without access control check
Principle of Fail-Safe Defaults

• Start by denying all access, then allow only that which has been explicitly permitted
  – oversights will then show up as “false negatives”
  • somebody is denied access that should be given it
  – opposite leads to “false positives”
  • somebody is given access that shouldn’t get it
  • bad guys usually don’t report this kind of failure…
• Examples:
  – Irix shipped with “xhost +” by default
    • Allows the world to open windows on your screen and grab the keystrokes you type

“Security through Obscurity” = bad

• Security through obscurity
  – attempting to gain security by hiding the implementation details of a system
  • claim: a secure system should be secure even if all implementation details are published
    • in fact, a system grows more secure as people scour over implementation details and find flaws
  • rely on mathematics and sound design to keep secure
• Counterexample: GSM cell phones
  – GSM committee designed their own crypto algorithm, but hid it from the world – “impossible to clone”
  • social engineering + reverse engineering revealed the algorithm
  • it turned out to be very weak
    – could essentially play questions with identity chip on cell phone, and eventually learn the secret key in a few hours

Trusted Computing Base (TCB)

• Think carefully about what you are trusting with your information
  – if you type your password on a keyboard, you’re trusting:
    • the keyboard manufacturer
    • your computer manufacturer
    • your operating system
    • the password library
    • the application that’s checking the password
  • how about typing your credit card number to a web service?
    • how about giving your credit card to a waiter?
• TCB = set of components (hardware, software, wetware) that you trust your secrets with
• Public web kiosks should *not* be in your TCB
  – should your OS? (IE and Active-X extensions)
  – ow about your compiler?
• A great read: "Reflections on Trusting Trust"

Authentication

• How does a computer know who I am?
  – user name / password
  • how does it store the password?
  • how does it check the password?
  • how secure is a password?
  – public/private keys
  – one-time keys
  – biometrics
• What does the computer do with this information?
  – assign you an identifier
    • UNIX: 32 bit number stored in process structure
    • Windows NT: 27 byte number, stored in an access token in kernel

Aside on Encryption

• Encryption: takes a key and data and creates ciphertext: \( E_k(M) = C \)
• Decryption: takes ciphertext and a key and recovers data: \( D_k(C) = M \)
• Symmetric algorithms (aka secret-key algorithms):
  – \( k_1 = k_2 \) (or can get \( k_2 \) from \( k_1 \))
• Public-Key Algorithms
  – decryption key (\( k_2 \)) cannot be calculated from encryption key (\( k_1 \))
  – encryption key can be made public
    • encryption key = “public key”, decryption key = “private key”
• Hashing: takes data and creates a fixed-size fingerprint, or hash
  – H(Attack at Dawn) = 183870
  – H(jIshack at dawn) = 465348
  – Can’t determine data from hash or find two pieces of data with same hash
Storing passwords

- CTSS (1962): password file [user name, user identifier, password]
  
  Bob, 14, “12.14.52”
  David, 15, “allison”
  Mary, 16, “t0f0t2n”

  If a bad guy gets hold of the password file, you’re in deep trouble!

- Unix (1974): encrypt passwords with passwords

  K=[0]allison

  Bob: 14: S6Uu0cYDVdTAk
  David: 15: J2ZI4ndBL6X.M
  Mary: 16: VW2bqvTa1BJKg

  David’s password, “allison,” is encrypted using itself as the key and stored in that form. Password can be checked by the system. No problem if someone steals the file – except for dictionary attacks

- Unix (1979): salted passwords

  Encryption is computed after affixing a number to the password. Thwarts pre-computed dictionary attacks

  K=[0]allison392

  Bob: 14: T7Vs1dZEWd8cl; 45
  David: 15: K3A5ocCM4ZM$; 392
  Mary: 16: WX3crwUnrnKLF; 152

Guessing passwords

- 26 letters used, 7 letters long
  - 8 billion passwords (33 bits)
  - Checking 100,000/second breaks in 22 hours
  - System should make checking passwords slow

- But most people’s passwords are not random sequences of letters!
  - girlfriend’s/boyfriend’s/spouse’s/dog’s name

- Dictionary attacks have traditionally been incredibly easy

Making it harder

- Using symbols and numbers and longer passwords
  - 95 characters, 14 characters long
  - $10^{27}$ passwords $\times$ 91 bits
  - Checking 100,000/second breaks in $10^{14}$ years

- Require frequent changing of passwords
  - guards against loaning it out, writing it down, etc.

Do longer passwords work?

- People can’t remember 14-character strings of random characters
- People write down difficult passwords
- People give out passwords to strangers
- Passwords can show up on disk
- If you are forced to change your password periodically, you probably choose an even dumber one
  - “feb04” “mar04” “apr04”
- How do we handle this in CSE?
Cool password attack

- VMS password checking flaw
  - password checking algorithm:
    ```c
    for (I=0; I<password.length(); I++) {
      if password[I] == supplied_password[I]
        return false;
    }
    return true;
    ```
  - can you see the problem?
  - another hint: think about virtual memory...
  - final hint: who controls where in memory supplied_password lives?

Login spoofers

- Login spoofers are a specialized class of Trojan horses
  - Can be circumvented by requiring an operation that
    unprivileged programs cannot perform
  - E.g. start login sequence with a key combination user
    programs cannot catch, CTRL+ALT+DEL on Windows

Sniffing passwords

- Incredibly, until just a couple of years ago we all
  entered cleartext passwords on the network!
  - including wireless LANs, where packet sniffing is duck soup!

Authorization

- How does the system know what I’m allowed to do?
  - logically, an authorization matrix:
    - objects = things that can be accessed
    - subjects/principals = things that can do the accessing (users or
      programs)

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carl</th>
</tr>
</thead>
<tbody>
<tr>
<td>etc</td>
<td>Read</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>homes</td>
<td>Read</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>/usr</td>
<td>None</td>
<td>None</td>
<td>Read</td>
</tr>
</tbody>
</table>

Modern security problems

- Confinement
  - How do I run code that I don’t trust?
    - e.g., RealPlayer, Flash
  - How do I restrict the data it can communicate?
  - What if trusted code has bugs?
    - e.g., Internet Explorer
  - Concept of "Least Privilege"
    - programs should only run with the minimal amount of privilege
      necessary

- Solutions
  - Restricted contexts – let the user divide their identity
  - ActiveX – make code writer identify self
  - Java – use a virtual machine that intercepts all calls
  - Binary rewriting – modify the program to force it to be safe
Restricted contexts
- Add extra identity information to a process
  - e.g., both username and program name (mikesw:navigator)
- Use both identities for access checks
  - add extra security checks at system calls that use program name
  - add extra ACLs on objects that grant/deny access to the program
- Allows users to sub-class themselves for less-trusted programs

ActiveX
- All code comes with a public-key signature
- Code indicates what privileges it needs
- Web browser verifies certificate
- Once verified, code is completely trusted

Java
- All problems are solved by a layer of indirection
  - All code runs on a virtual machine
  - Virtual machine tracks security permissions
  - Allows fancier access control models - allows stack walking
- JVM doesn’t work for other languages
- Virtual machines can be used with all languages
  - Run virtual machine for hardware
  - Inspect stack to determine subject for access checks

Binary rewriting
- Goal: enforce code safety by embedding checks in the code
- Solution:
  - Compute a mask of accessible addresses
  - Replace system calls with calls to special code

Attacks: Trojan Horses
- A malicious program disguised as an innocent one
- Login spoofers are a specialized class of Trojan horses
  - Can be circumvented by requiring an operation that unprivileged programs cannot perform
  - E.g. Start login sequence with a key combination user programs cannot catch, CTRL+ALT+DEL on Windows

Attacks: Viruses and Worms
- Viruses: passive code attached to other programs
  - E.g. a program that modifies MS Word
- Worms: code that actively replicates itself and does not depend on the execution of another program to spread
  - E.g. the Internet worm
- Buffer overflow
  - C string libraries hard to use correctly
    - e.g. easy to write outside string bounds
  - Most OS code is written in C, many systems have vulnerabilities
    - If a string is stored on the stack, someone can modify the behavior of a program by going off the end of the string and changing a return address stored on stack
Attacks: Denial of service

- Attacker sends legitimate-looking requests for service to a service provider
- Service provider commits the necessary resources to provide the service
  - Ports, buffer space, bandwidth
- The resources are wasted, legitimate users get diminished service
  - Usually launched from many computers controlled by attackers
- Possible whenever the cost to ask for service is far cheaper than the cost of providing it
  - Challenge-response mechanism