Software and Security

Objectives

- To present importance of security at system level
- To define and discuss components of the systems involved and level of security associated with each of them
- To provide overview of malicious programs
- To describe commonly known malicious programs like virus, worm, Trojans, logic bombs etc.
- To present an overview of IDS
- To discuss firewalls and their classifications

System

- Comprises of computing and communication environment over which developers have some control
- System components
  - Security relevant- crucial components to which malfunction or penetration can lead to security violations.
    - E.g. OS and computer hardware examples
  - Others- Objects that system controls and protects
    - Programs (not processes), data, terminal, modem
- Security perimeter- line of demarcation between security relevant and other components

User, trust and trusted systems

- User- a person whose information system protects and whose access to information is controlled by system
- User is trusted with some confidential information.
- System security needs to have trust in security related components inside the security perimeter.
- Trust in systems is built using techniques of identification and authentication.

Why Software?

- Why is software as important to security as crypto, access control and protocols?
- Virtually all of information security is implemented in software
- If your software is subject to attack, your security is broken
  - Regardless of strength of crypto, access control or protocols
- Software is a poor foundation for security

Software Issues

"Normal" users

- Find bugs and flaws by accident
- Hate bad software…
- …but must learn to live with it
- Must make bad software work

Attackers

- Actively look for bugs and flaws
- Like bad software…
- …and try to make it misbehave
- Attack systems thru bad software

Complexity

- "Complexity is the enemy of security", Paul Kocher, Cryptography Research, Inc.

system	Lines of code (LOC)
Netscape	17,000,000
Space shuttle	10,000,000
Linux	1,500,000
Windows XP	40,000,000
Boeing 777	7,000,000

- A new car contains more LOC than was required to land the Apollo astronauts on the moon

Software Security Topics

- Program flaws (unintentional)
  - Buffer overflow
  - Incomplete mediation
  - Race conditions
- Malicious software (intentional)
  - Viruses
  - Worms
  - Other breeds of malware

Example

```
char array[10];
for(i = 0; i < 10; ++i)
        array[i] = `A`;
array[10] = `B`;
```

- This program has an **error**
- This error might cause a **fault**
  - Incorrect internal state
- If a fault occurs, it might lead to a **failure**
  - Program behaves incorrectly (external)
- We use the term **flaw** for all of the above

Secure Software

- In software engineering, try to insure that a program does what is intended
- Secure software engineering requires that the software **does what is intended…**
- **…and nothing more**
- Absolutely secure software is impossible
  - Absolute security is almost never possible!
- How can we manage the risks?

Program Flaws

- Program flaws are unintentional
  - But still create security risks
- We'll consider 3 types of flaws
  - Buffer overflow (smashing the stack)
  - Incomplete mediation
  - Race conditions
- Many other flaws can occur
- These are most common

Buffer Overflow

Typical Attack Scenario

- Users enter data into a Web form
- Web form is sent to server
- Server writes data to buffer, without checking length of input data
- Data overflows from buffer
- Sometimes, overflow can enable an attack
- Web form attack could be carried out by anyone with an Internet connection

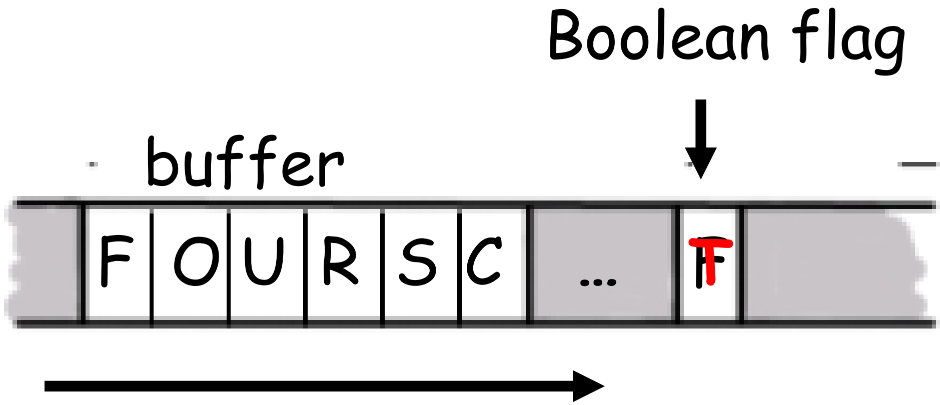
Buffer Overflow

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

- **Q:** What happens when this is executed?
- **A:** Depending on what resides in memory at location "buffer[20]"
  - Might overwrite **user** data or code
  - Might overwrite **system** data or code

Simple Buffer Overflow

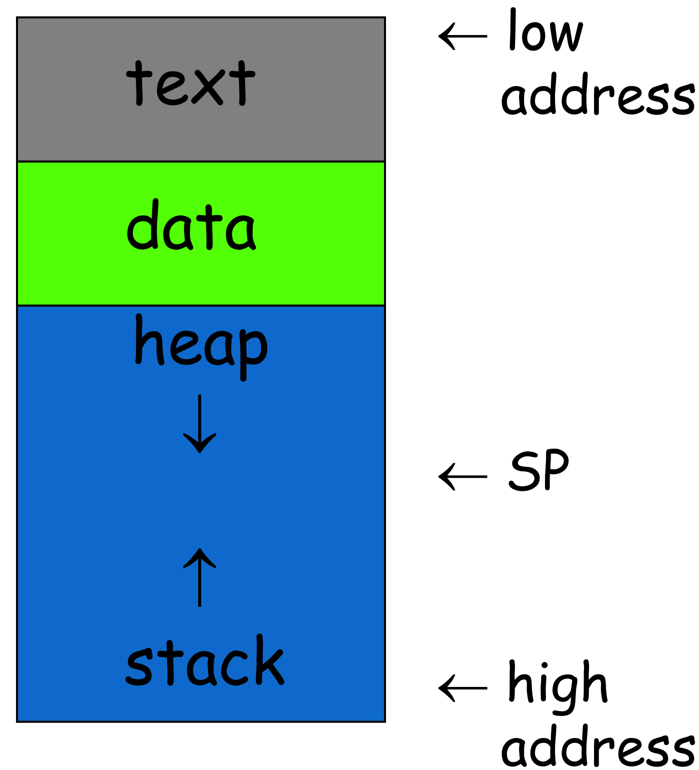
- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate!



- In some cases, attacker need not be so lucky as to have overflow overwrite flag

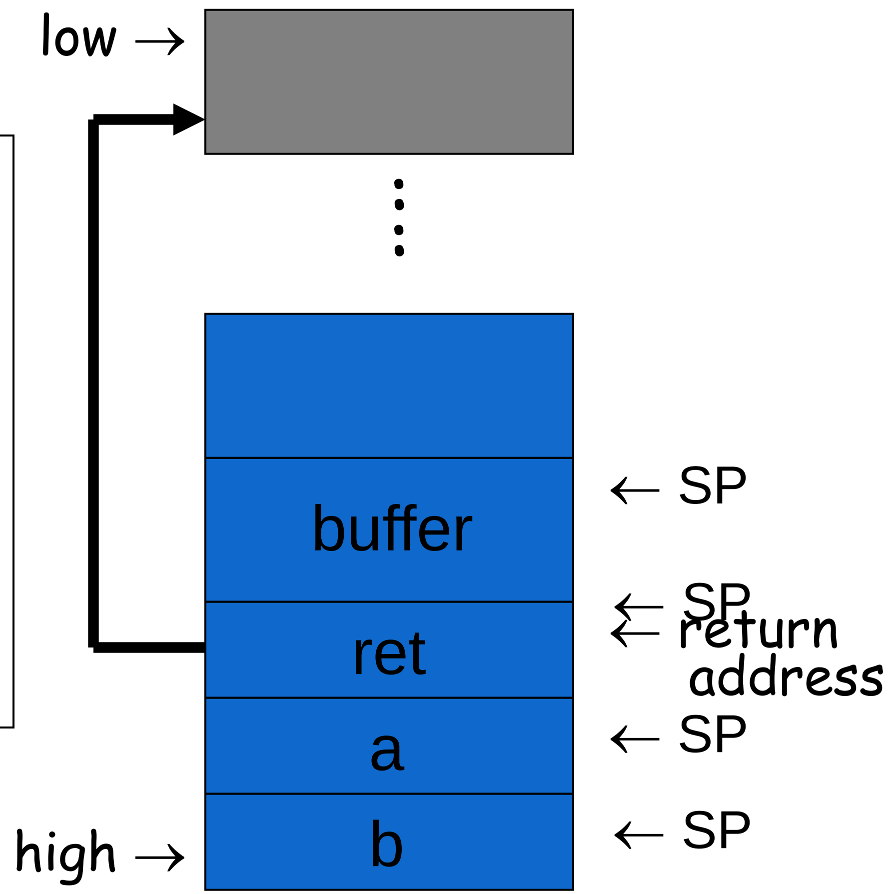
Memory Organization

- **Text** == code
- **Data** == static variables
- **Heap** == dynamic data
- **Stack** == "scratch paper"
  - Dynamic local variables
  - Parameters to functions
  - Return address



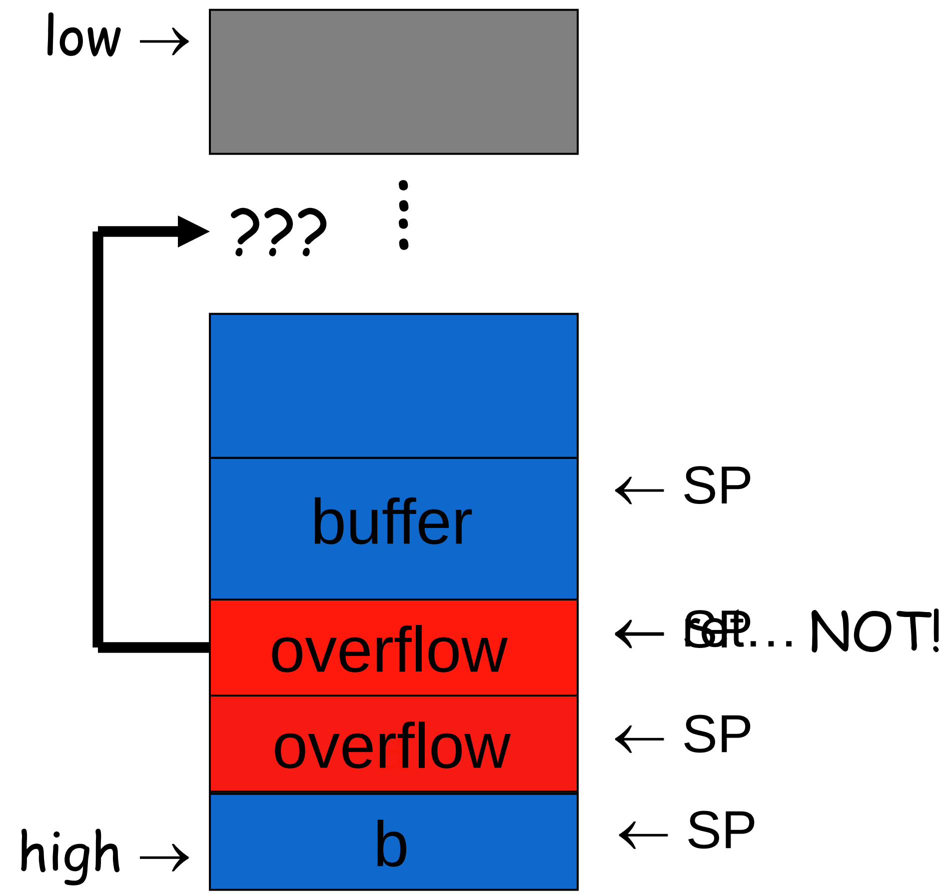
Simplified Stack Example

```
void func(int a, int b){
    char buffer[10];
}
void main(){
    func(1, 2);
}
```



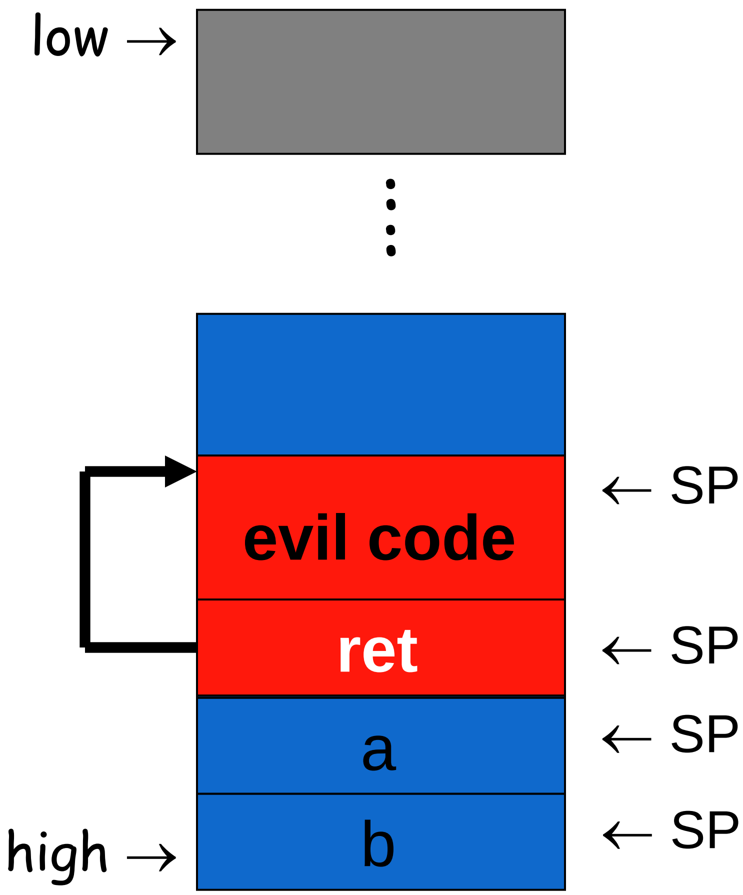
Smashing the Stack

- What happens if buffer overflows?
- Program "returns" to wrong location
- A crash is likely



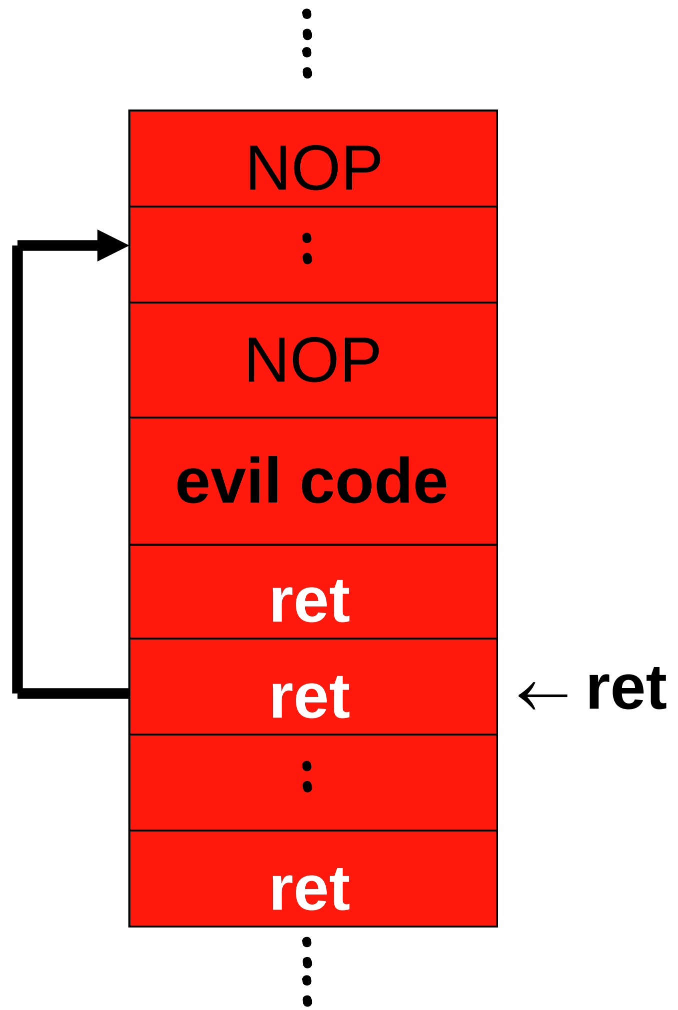
Smashing the Stack

- Trudy has a better idea…
- **Code injection**
- Trudy can run code of her choosing!



Smashing the Stack

- Trudy may not know
  - Address of evil code
  - Location of ret on stack
- Solutions
  - Precede evil code with NOP "landing pad"
  - Insert lots of new **ret**



Stack Smashing Summary

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must line up just right
- If exploitable, attacker can **inject code**
- Trial and error likely required
  - Lots of help available online
  - Smashing the Stack for Fun and Profit, Aleph One
- Also heap overflow, integer overflow, etc.
- Stack smashing is "attack of the decade"

Stack Smashing Example

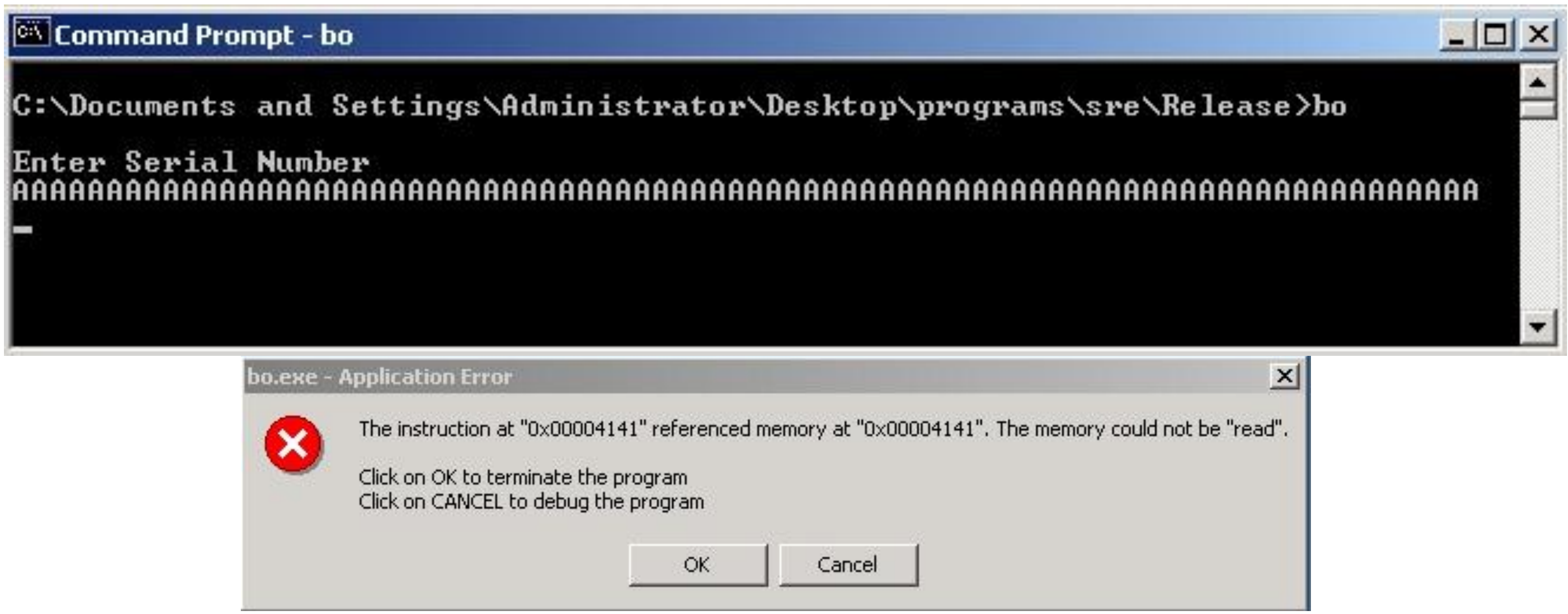
- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)

```
Command Prompt

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
woeiweiow

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

- Program quits on incorrect serial number

Example

- By trial and error, attacker discovers an apparent buffer overflow



- Note that 0x41 is "A"
- Looks like **ret** overwritten by 2 bytes!

Example

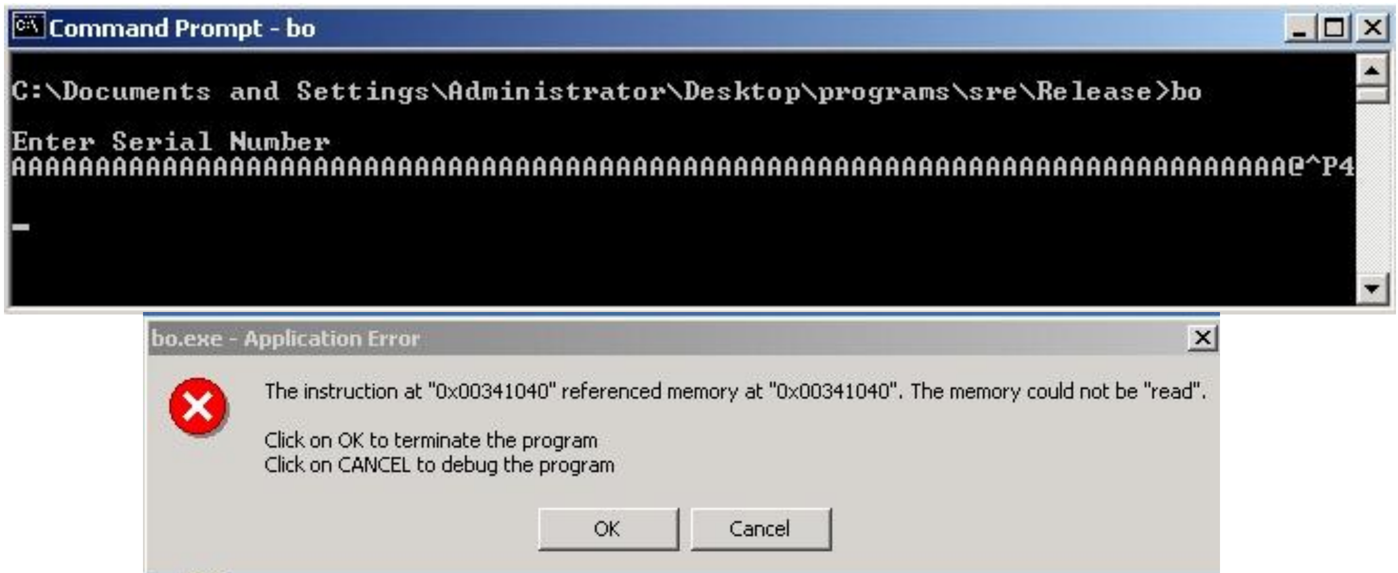
- Next, disassemble bo.exe to find

```
.text:00401000
.text:00401000                 sub     esp, 1Ch
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_40109F
.text:0040100D                 lea     eax, [esp+20h+var_1C]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401088
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+2Ch+var_1C]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401050
.text:0040102D                 add     esp, 18h
.text:00401030                 test    eax, eax
.text:00401032                 jnz     short loc_401041
.text:00401034                 push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039                 call    sub_40109F
.text:0040103E                 add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Example

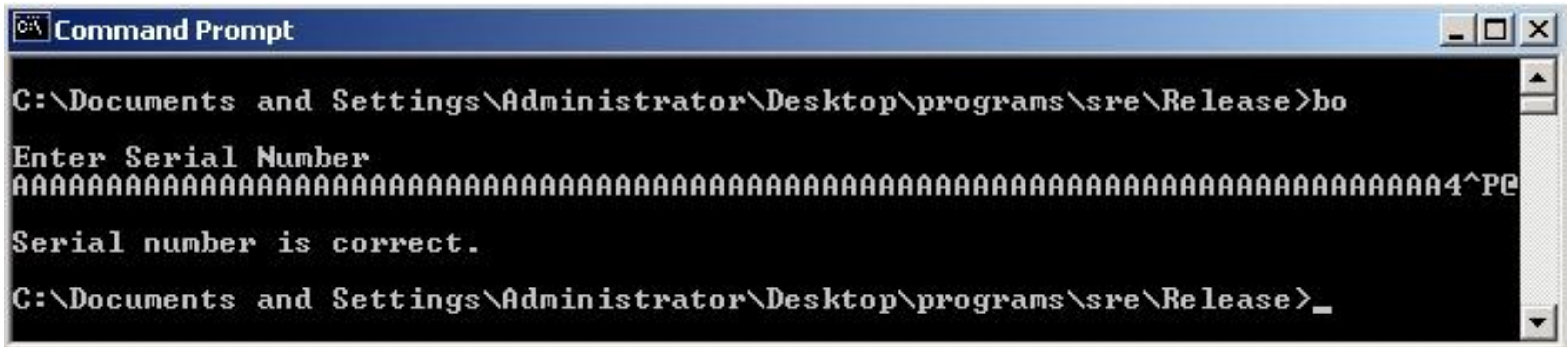
- Find that 0x401034 is "@^P4" in ASCII



- Byte order is reversed? Why?
- X86 processors are "little-endian"

Example

- Reverse the byte order to "4^P@" and…



- Success! We've bypassed serial number check by exploiting a buffer overflow
- Overwrote the return address on the stack

Example

- Attacker did not require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Can find address by trial and error
  - Necessary if attacker does not have exe
  - For example, a remote attack

Example

- Source code of the buffer overflow
- Flaw easily found by attacker
- **Even without the source code!**

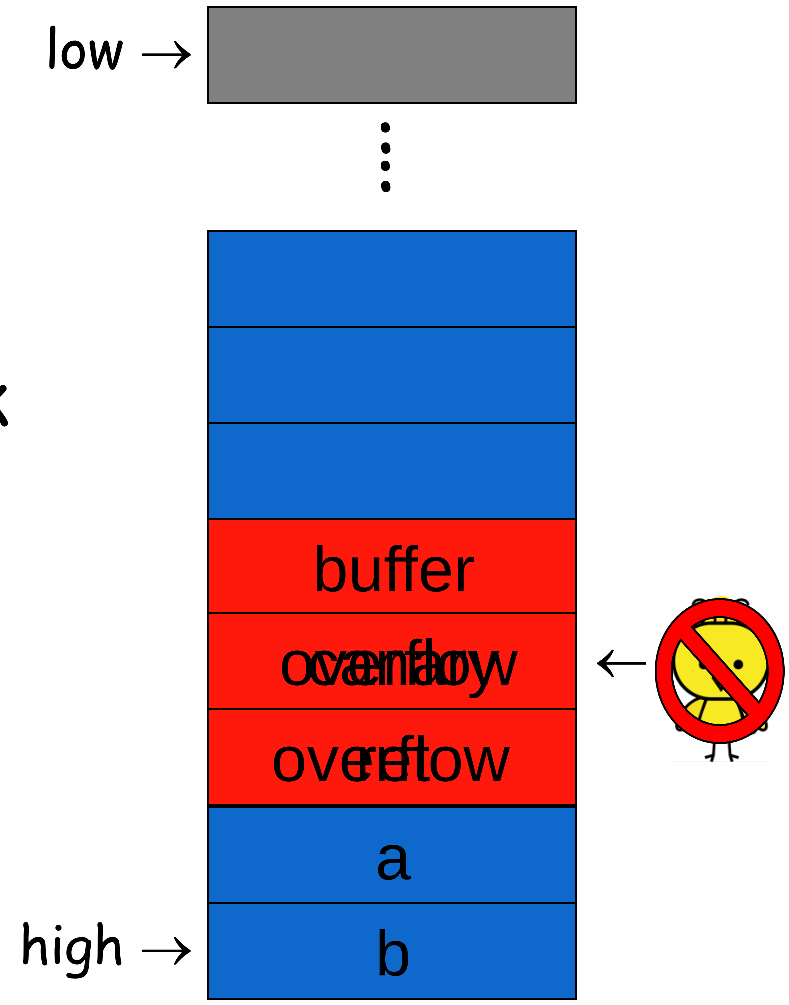
```
#include <stdio.h>
#include <string.h>

main()
{
    char in[75];

    printf("\nEnter Serial Number\n");

    scanf("%s", in);

    if(!strncmp(in, "S123N456", 8))
    {
       printf("Serial number is correct.\n");
    }
}
```

Stack Smashing Prevention

- 1st choice: employ **non-executable stack**
  - "No execute" **NX bit** (if available)
  - Seems like the logical thing to do, but some real code executes on the stack (Java does this)
- 2nd choice: use **safe languages** (Java, C#)
- 3rd choice: use **safer C functions**
  - For unsafe functions, there are safer versions
  - For example, strncpy instead of strcpy

Stack Smashing Prevention

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or value depends on **ret**



Microsoft's Canary

- Microsoft added **buffer security check** feature to C++ with /GS compiler flag
- Uses canary (or "security cookie")
- **Q:** What to do when canary dies?
- ***A:*** Check for user-supplied handler
- Handler may be subject to attack
  - Claimed that attacker can specify handler code
  - If so, "safe" buffer overflows become exploitable when /GS is used!

Buffer Overflow

- The "attack of the decade" for 90's
- Will be the attack of the decade for 00's
- Can be prevented
  - Use safe languages/safe functions
  - Educate developers, use tools, etc.
- Buffer overflows will exist for a long time
  - Legacy code
  - Bad software development

Incomplete Mediation

Input Validation

- Consider: strcpy(buffer, argv[1])
- A buffer overflow occurs if

  len(buffer) < len(argv[1])
- Software must **validate** the input by checking the length of argv[1]
- Failure to do so is an example of a more general problem: **incomplete mediation**

Input Validation

- Consider web form data
- Suppose input is validated on client
- For example, the following is valid

  http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=205

- Suppose input is not checked on server
  - Why bother since input checked on client?
  - Then attacker could send http message

  http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=25

Incomplete Mediation

- Linux kernel
  - Research has revealed many buffer overflows
  - Many of these are due to incomplete mediation
- Linux kernel is "good" software since
  - Open-source
  - Kernel — written by coding gurus
- Tools exist to help find such problems
  - But incomplete mediation errors can be subtle
  - And tools useful to attackers too!

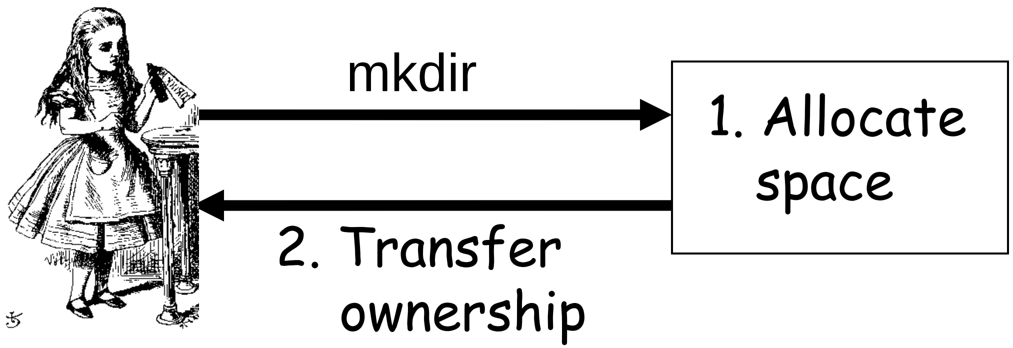
Race Conditions

Race Condition

- Security processes should be **atomic**
  - Occur "all at once"
- Race conditions can arise when security-critical process occurs in stages
- Attacker makes change between stages
  - Often, between stage that gives authorization, but before stage that transfers ownership
- Example: Unix mkdir

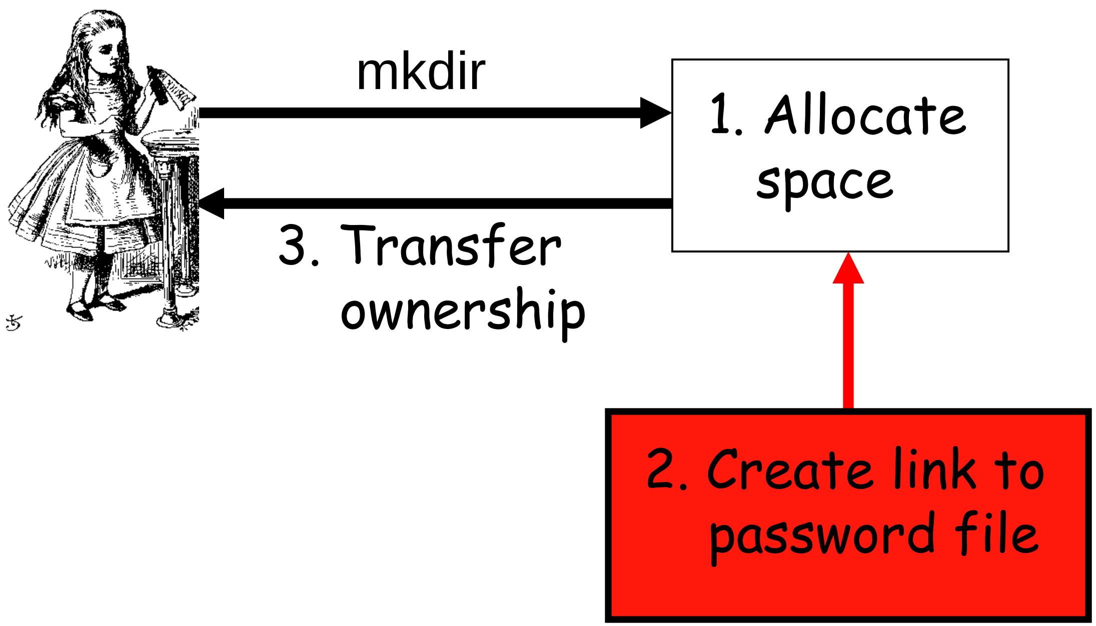
mkdir Race Condition

- mkdir creates new directory
- How mkdir is supposed to work



mkdir Attack

- The mkdir **race condition**



- Not really a "race"
  - But attacker's timing is critical

Race Conditions

- Race conditions are common
- Race conditions may be more prevalent than buffer overflows
- But race conditions harder to exploit
  - Buffer overflow is "low hanging fruit" today
- To prevent race conditions, make security-critical processes atomic
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice

Malware

Malicious software

- Programs which try to subvert expected operation of secured and benign codes
- Most common categories-
  - Worms
  - Viruses
  - Logic bombs
  - Trojans
  - Spyware
  - adware

Malicious Software

- Malware is not new…
- Fred Cohen's initial virus work in 1980's
  - Used viruses to break MLS systems
- Types of malware (lots of overlap)
  - **Virus** — passive propagation
  - **Worm** — active propagation
  - Trojan horse — unexpected functionality
  - Trapdoor/backdoor — unauthorized access
  - Rabbit — exhaust system resources

Worms

- Run independently
- Propagate a full working version of itself to other machines
- Analogous to parasites which live inside a host and use its resources for its existence
- Classified by primary method they use for transport
  - IM Worms
  - Email worms

Virus

- Cannot run independently
- Need host program to run and activate them
- A computer virus has-
  - Infection mechanism
  - Payload
  - Trigger

Virus pseudocode

```
infect();
if trigger( )
  then payload();
```

Where do Viruses Live?

- Just about anywhere…
- Boot sector
  - Take control before anything else
- Memory resident
  - Stays in memory
- Applications, macros, data, etc.
- Library routines
- Compilers, debuggers, virus checker, etc.
  - These are particularly nasty!

Virus classification by target

- Boot sector virus
  - Primary boot
  - Secondary boot
- Executable file infectors
  - Prepending Virus -placed at beginning,
  - Appending virus- placed at end,
  - Virus code is over-written or inserted into a file
- Data file infectors- macro virus

Virus classification by target

- Overwriting virus
  - Do not change target file size
- Companion virus
  - Do not modify infected code
  - Installs itself in such a way that it gets executed before the target code

Virus classification based on concealment

- Encryption
- Oligomorphism
- Polymorphism
- Metamorphism

Virus classification - Encryption

- Makes detection difficult
- Has a decryptor loop for decryption and transfer of control to it
- Encryption techniques used
  - Simple transformation
  - Key mixing
  - Substitution cipher
  - Strong encryption
- Signature detection is easy

Virus classification - Oligomorphism

- uses a pool of decryptors Instead of one; so uses varying keys
- Entire virus changes and becomes harder to detect
- Difficulty is very marginal as anti-virus needs to check only loop variants

Virus classification - Polymorphism

- Almost same as Oligomorphism but has extremely large number of decryptor loops
- Mutation engine changes loop with every encryption

Methods used for writing viruses

- Instruction equivalence
- Instruction sequence equivalence
- Instruction reordering
- Register renaming
- Concurrency
- Writing convoluted programs
- Inlining & outlining function calls

Virus classification - Metamorphism

- Do not have decryption loops
- Mutation engine changes for every infection

Logic bombs

- Has typically two parts
  - Payload-malicious piece of code
  - Trigger- Boolean logic
- Time bombs are examples of logic bombs

Trojans

- Malicious programs that perform some harmless activities in addition to malicious activities

Trojan Horse Example

- A trojan has unexpected function
- Prototype of trojan for the Mac
- File icon for freeMusic.mp3:

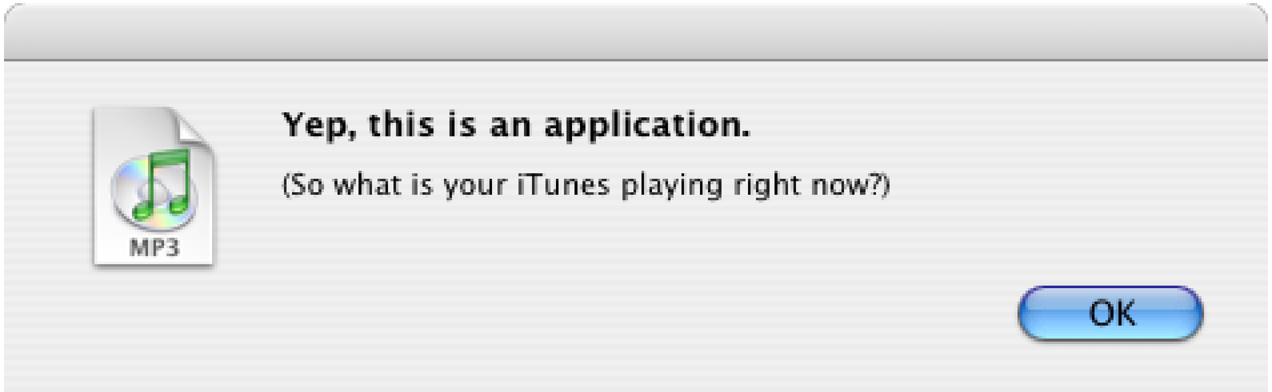
freeMusic.mp3

- For a real mp3, double click on icon
  - iTunes opens
  - Music in mp3 file plays
- But for freeMusic.mp3, unexpected results…

Mark Stamp's Information Security: Principles and Practices by Mark Stamp/ Deven Shah.

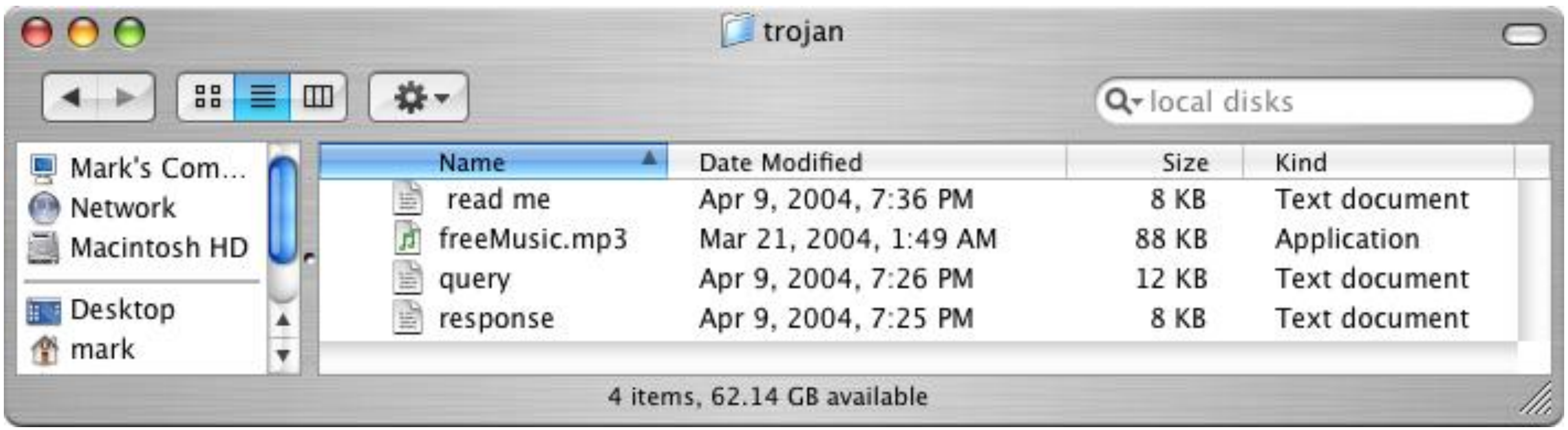
Trojan Example

- Double click on freeMusic.mp3
  - iTunes opens (expected)
  - "Wild Laugh" (probably not expected)
  - Message box (unexpected)



Trojan Example

- How does freeMusic.mp3 trojan work?
- This "mp3" is an application, not data!



- This trojan is harmless, but...
- Could have done anything user can do
  - Delete files, download files, launch apps, etc.

Spyware

- A software used to collect & transmit information from victim computer
- Spywares do not replicate themselves
- Different form of trojans
- Often get downloaded when viewing some webpage, called drive by download concept
- Examples of info gathered by spywares
  - Passwords
  - Credit card numbers and bank secrets
  - Software license keys

Adwares

- Have similarities with spywares
- Not self-replicating
- Objective is marketing

Malware Detection

- Three common methods
  - Signature detection
  - Change detection
  - Anomaly detection
- We'll briefly discuss each of these
  - And consider advantages and disadvantages of each

Signature Detection

- A **signature** is a string of bits found in software (or could be a hash value)
- Suppose that a virus has signature 0x23956a58bd910345
- We can search for this signature in all files
- If we find the signature are we sure we've found the virus?
  - No, same signature could appear in other files
  - But at random, chance is very small: $1/2^{64}$
  - Software is not random, so probability is higher

Signature Detection

- Advantages
  - Effective on "traditional" malware
  - Minimal burden for users/administrators
- Disadvantages
  - Signature file can be large (10,000's)…
  - …making scanning slow
  - Signature files must be kept up to date
  - Cannot detect unknown viruses
  - Cannot detect some new types of malware
- By far the most popular detection method

Change Detection

- Viruses must live somewhere on system
- If we detect that a file has changed, it may be infected
- How to detect changes?
  - Hash files and (securely) store hash values
  - Recompute hashes and compare
  - If hash value changes, file **might** be infected
  - Check for oligomorphism and polymorphism

Change Detection

- Advantages
  - Virtually no false negatives
  - Can even detect previously unknown malware
- Disadvantages
  - Many files change — and often
  - Many false alarms (false positives)
  - Heavy burden on users/administrators
  - If suspicious change detected, then what?
  - Might still need signature-based system

Anomaly Detection

- Monitor system for anything "unusual" or "virus-like" or potentially malicious
- What is unusual?
  - Files change in some unusual way
  - System misbehaves in some way
  - Unusual network activity
  - Unusual file access, etc., etc., etc.
- But must first define "normal"
  - And normal can change!

Anomaly Detection

- Advantages
  - Chance of detecting unknown malware
- Disadvantages
  - Unproven in practice
  - Trudy can make abnormal look normal (go slow)
  - Must be combined with another method (such as signature detection)
- Also popular in intrusion detection (IDS)
- A difficult unsolved (unsolvable?) problem
  - As difficult as AI?

Not in syllabus- Given for information
Miscellaneous Attacks

Miscellaneous Attacks

- Numerous attacks involve software
- We'll discuss a few issues that do not fit in previous categories
  - Salami attack
  - Linearization attack
  - Time bomb
  - Can you ever trust software?

Salami Attack

- What is Salami attack?
  - Programmer "slices off" money
  - Slices are hard for victim to detect
- Example
  - Bank calculates interest on accounts
  - Programmer "slices off" any fraction of a cent and puts it in his own account
  - No customer notices missing partial cent
  - Bank may not notice any problem
  - Over time, programmer makes lots of money!

Salami Attack

- Such attacks are possible for insiders
- Do salami attacks actually occur?
- Programmer added a few cents to every employee payroll tax withholding
  - But money credited to programmer's tax
  - Programmer got a big tax refund!
- Rent-a-car franchise in Florida inflated gas tank capacity to overcharge customers

Salami Attacks

- Employee reprogrammed Taco Bell cash register: $2.99 item registered as $0.01
  - Employee pocketed $2.98 on each such item
  - A large "slice" of salami!
- In LA four men installed computer chip that overstated amount of gas pumped
  - Customer complained when they had to pay for more gas than tank could hold!
  - Hard to detect since chip programmed to give correct amount when 5 or 10 gallons purchased
  - Inspector usually asked for 5 or 10 gallons!

Linearization Attack

- Program checks for serial number S123N456
- For efficiency, check made one character at a time
- Can attacker take advantage of this?

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int i;
    char serial[9]="S123N456\n";

    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i]) break;
    }
    if(i == 8)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

Linearization Attack

- Correct string takes longer than incorrect
- Attacker tries all 1 character strings
  - Finds S takes most time
- Attacker then tries all 2 char strings S*
  - Finds S1 takes most time
- And so on…
- Attacker is able to recover serial number one character at a time!

Linearization Attack

- What is the advantage of attacking serial number one character at a time?
- Suppose serial number is 8 characters and each has 128 possible values
  - Then $128^8 = 2^{56}$ possible serial numbers
  - Attacker would guess the serial number in about 2^{55} tries — a lot of work!
  - Using the linearization attack, the work is about $8*(128/2) = 2^9$ which is trivial!

Linearization Attack

- A real-world linearization attack
- TENEX (an ancient timeshare system)
  - Passwords checked one character at a time
  - Careful timing was not necessary, instead…
  - …could arrange for a "page fault" when next unknown character guessed correctly
  - The page fault register was user accessible
  - Attack was very easy in practice

Time Bomb

- In 1986 Donald Gene Burleson told employer to stop withholding taxes from his paycheck
- His company refused
- He planned to sue his company
  - He used company computer to prepare legal docs
  - Company found out and fired him
- Burleson had been working on a malware…
- After being fired, his software "time bomb" deleted important company data

Time Bomb

- Company was reluctant to pursue the case
- So Burleson sued company for back pay!
  - Then company finally sued Burleson
- In 1988 Burleson fined $11,800
  - Took years to prosecute
  - Cost thousands of dollars to prosecute
  - Resulted in a slap on the wrist
- One of the first computer crime cases
- Many cases since follow a similar pattern
  - Companies often reluctant to prosecute

Thank You