Chapter 9:

Security

Textbook Reading:

... Sections to be covered?...

Protection & Security

A major concern of OS's

More complex OS's --> System bloat --> More problems!!!

Data Confidentiality:

Keeping secret data from being exposed

Data Integrity:

Unauthorized users must not be able to change / delete info

System Availability:

Guard against "denial of service" attacks

Types of Bad Guys

"intruders," "adversaries"

Casual prying by non-technical users

Simple barriers will work

Snooping by insiders

A personal challenge to every CS student

Attempts to make money

The Bank Interest Scheme:

Accounts get interest

Idea: truncate, rather than round

Blackmail: "Pay me or else your files will be deleted"

Commercial / Military Espionage

Well-funded, sophisticated, high-tech

Wire-tapping, specialized antennas, bribery, lots of CPU

Malicious Software (e.g., viruses)

Like terrorism: destroys at random

Accidental Loss

Protect against:

- **"Acts of God"**
  fires, floods, earthquakes, civil unrest, ...

- **Hardware Errors**
  Disk crash, transmission errors

- **Software Errors**
  Bugs

- **Human Errors**
  Type the wrong thing (`rm *`)
  Forget to do a backup (or "restore" instead of "backup")

Cryptography

Plaintext (P)

The material that should be protected

Can be reduced to a series of numbers

Ciphertext (C)

The encoded version

Encryption Algorithm (E)

Plaintext --> Ciphertext

Decryption Algorithm (D)

Ciphertext --> Plaintext

Cryptography

Cryptography

Encryption:

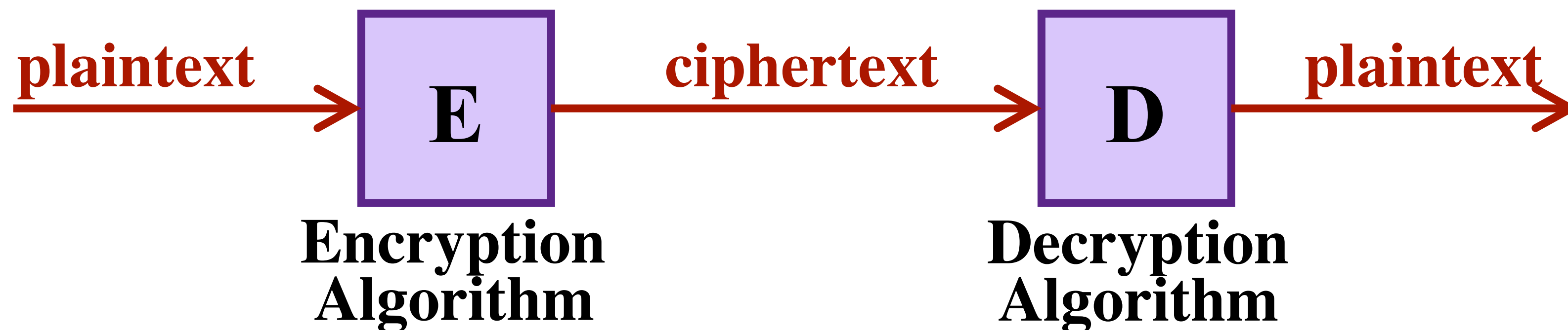
$C = E(P)$

Decryption

$P = D(C)$

Must have:

$P = D(E(P))$



Cryptography

Encryption:

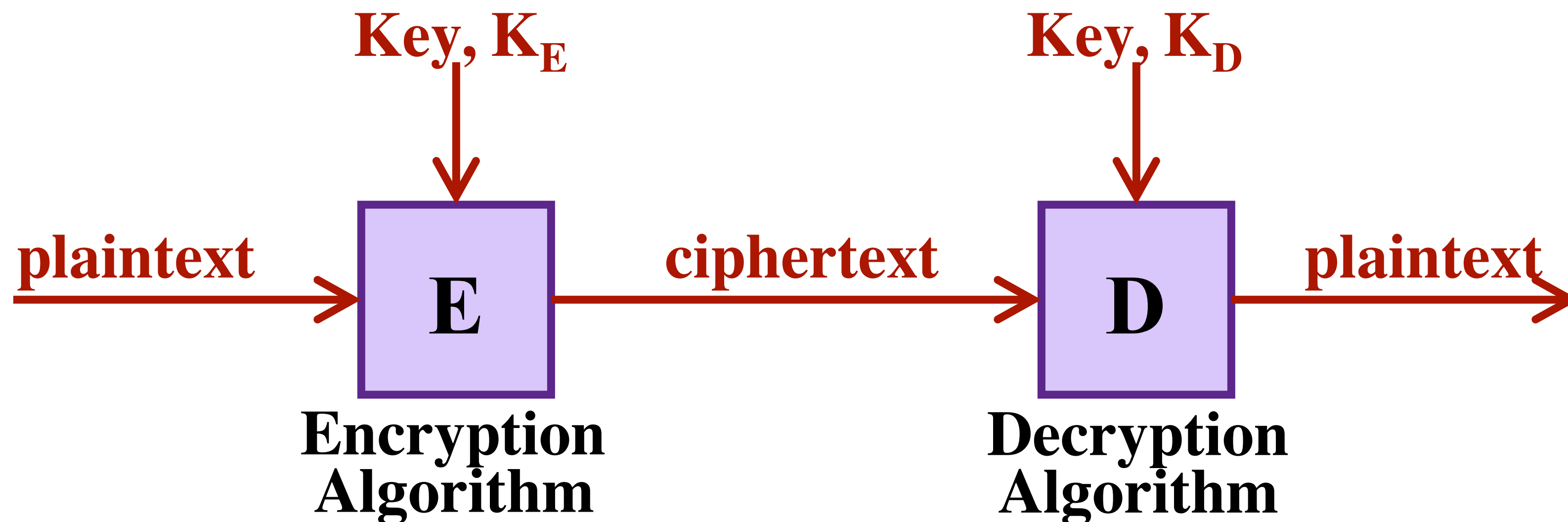
$C = E(P)$

Decryption

$P = D(C)$

Must have:

$P = D(E(P))$



Cryptography

Encryption:

$C = E(P)$

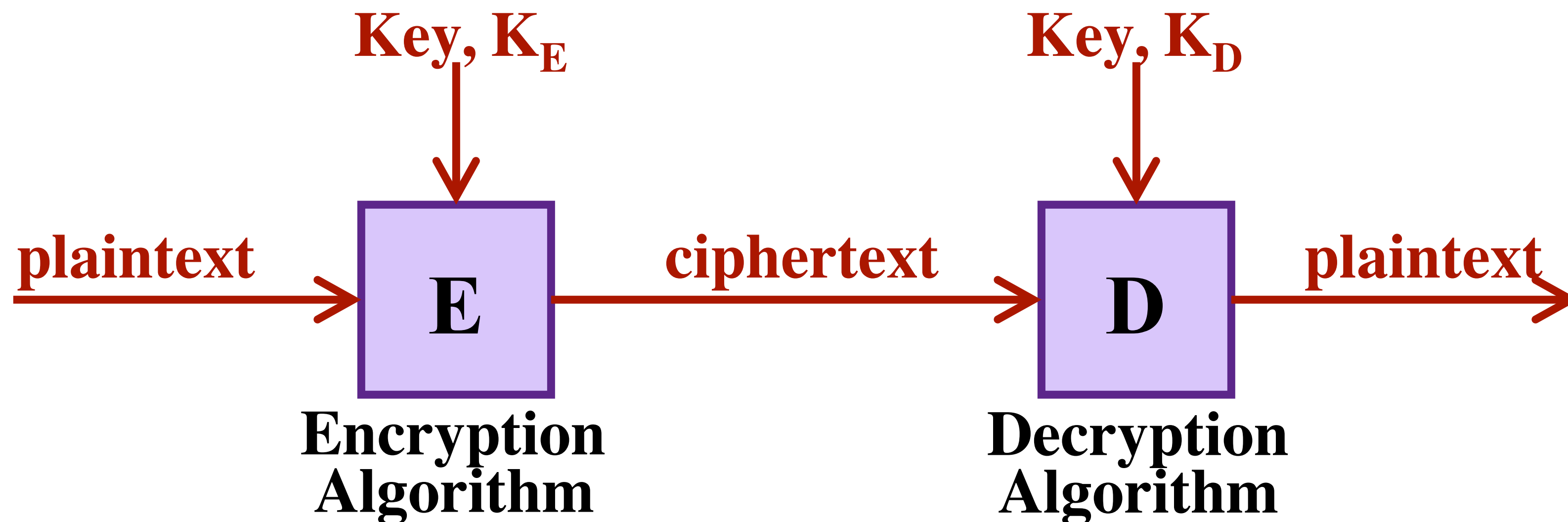
Decryption

$P = D(C)$

Must have:

$P = D(E(P))$

Want a single (good) algorithm parameterized with a key



Cryptography

Encryption:

$C = E(P, K_E)$

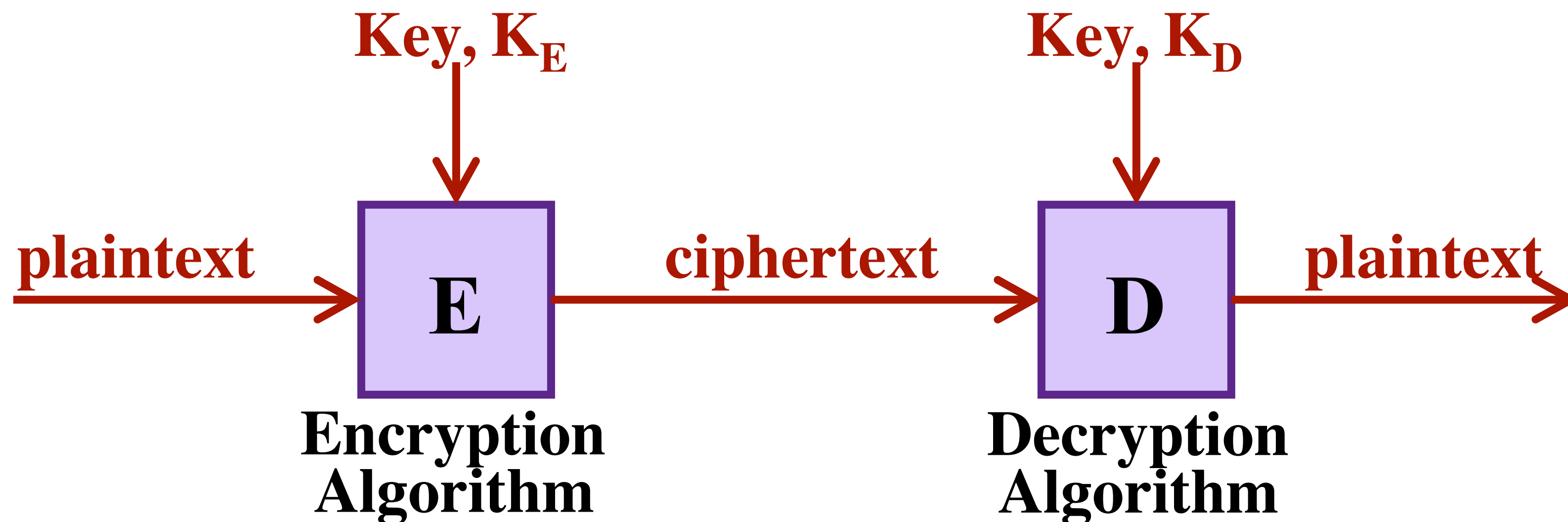
Decryption

$P = D(C, K_D)$

Must have:

$P = D(E(P, K_E), K_D)$

Want a single (good) algorithm parameterized with a key



Cryptography

"Symmetric-key Cryptography"
Also called "Secret-key Cryptography"

If you have one key, it is easy to find the other.
Effectively: There is only one key.

With key size of 1000 bits, this can be secure.

Problem:
The sender and receiver share the key.
Communicating the key (*securely*) is a nuisance!

Public-Key Cryptography

The encryption key is different from the decryption key.

Uses encyption / decryption algorithms such that...
Given encryption key,
it is virtually impossible to discover the decryption key.

Public-Key Cryptography

The encryption key is different from the decryption key.

Uses encyption / decryption algorithms such that...
Given encryption key,
it is virtually impossible to discover the decryption key.

Example:
Squaring and square-rooting are inverse operations.
... but one is much more difficult than the other!
What is 738×738 ? **(easy)**
What is $\sqrt{544{,}644}$? **(hard)**

Encryption uses the easy operation.
Decryption uses the hard operation.

Public-Key Cryptography

The encryption key is different from the decryption key.

Uses encyption / decryption algorithms such that...
Given encryption key,
it is virtually impossible to discover the decryption key.

Example:
Squaring and square-rooting are inverse operations.
... but one is much more difficult than the other!
What is 738×738 ? (easy)
What is $\sqrt{544{,}644}$? (hard)

Encryption uses the easy operation.
Decryption uses the hard operation.

"RSA Public-Key Encryption" uses this principle.

Public-Key Cryptography

Each person picks their own pair of keys

`< public-key, private-key >`

(This part is automated.)

The public key is for encryption.
The private key is for decryption.

They publish their public-key for everyone to see.
They keep their private key secret.

Public-Key Cryptography

Each person picks their own pair of keys

`< public-key, private-key >`

(This part is automated.)

The public key is for encryption.
The private key is for decryption.

They publish their public-key for everyone to see.
They keep their private key secret.

To send someone a message...

Obtain the receiver's public key.
Encrypt the message with their public key.
No-one but the intended receiver can decrypt it!

Digital Signatures

Scenario:

Assume persons X and Y want to agree to a contract
...via the Internet / email
They exchange emails saying "I agree to such-and-such".
Later, a legal conflict arises.
X claims that he never agreed to anything.
X claims the email is a forgery.

Problem:

How can Y prove that X really "signed" the agreement?

Goal:

Send an email with a "signature" that cannot be forged or repudiated.

Digital Signatures

Given a body of text...

(e.g., the email contract to be "signed")

A One-Way Hash

Input: The text (i.e., many, many bytes of data)

Output: A single number

...such that any change to the text would result in a different number

Examples:

MD5 (Message Digest)

Produces a 16-byte result

SHA (Secure Hash Algorithm)

Produces a 20-byte result

Digital Signatures

Assume:

We are using Public-Key Encryption

... and that it is *commutative*

Doesn’t matter if you encrypt, then decrypt

or decrypt, then encrypt...

Digital Signatures

Assume:

We are using Public-Key Encryption

... and that it is *commutative*

Doesn't matter if you encrypt, then decrypt

or decrypt, then encrypt...

Recall:

$$P = D(E(P, K_E), K_D)$$

Also want:

$$P = E(D(P, K_D), K_E)$$

Digital Signatures

Assume:

We are using Public-Key Encryption

... and that it is *commutative*

Doesn't matter if you encrypt, then decrypt

or decrypt, then encrypt...

public

private

Recall:

$P = D(E(P, K_E), K_D)$

private

public

Also want:

$P = E(D(P, K_D), K_E)$

The RSA public-key encryption algorithm has this property.

Digital Signatures

The sender...

Composes a plaintext document

Doc

Computes its hash value

Hash (Doc)

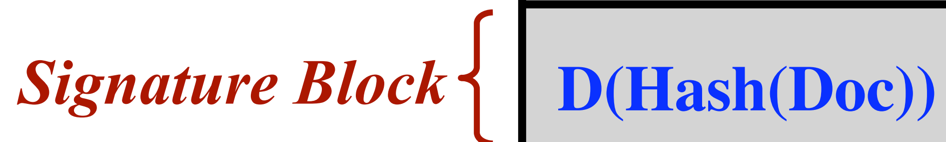
Decrypts with private key

D (Hash (Doc))

Transmits

< Doc, D (Hash (Doc)) >

Plaintext
Doc



Digital Signatures

The receiver...

Receives the message

< Doc, D (Hash (Doc)) >

Computes hash of Document

Hash (Doc)

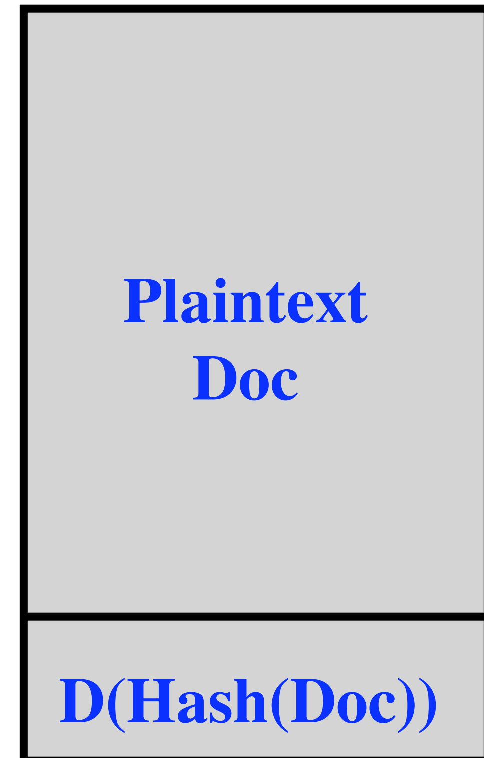
Encrypts the signature block

E (D (Hash (Doc)))

Compares the two

Hash (Doc)

E (D (Hash (Doc)))



Digital Signatures

The receiver...

Receives the message

< Doc, D (Hash (Doc)) >

Computes hash of Document

Hash (Doc)

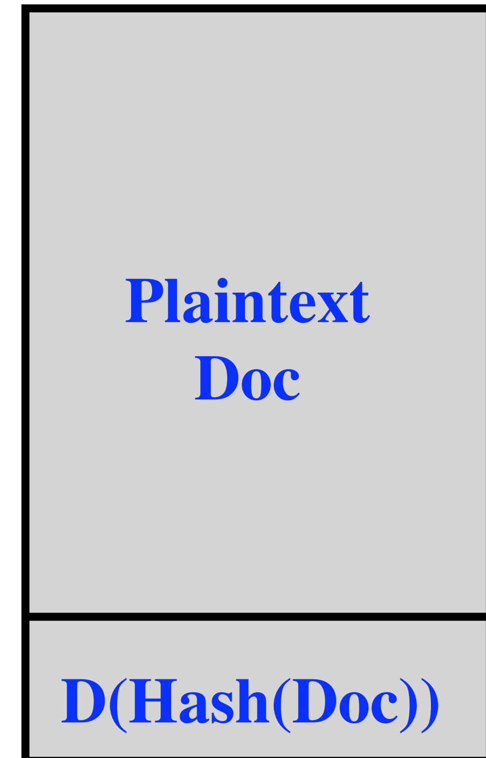
Encrypts the signature block

E (D (Hash (Doc)))

Compares the two

Hash (Doc)

E (D (Hash (Doc)))



If equal...

- *Only the owner of the private key could have created this signature block.*
- *The document has not been altered since signing.*

Digital Certificates

Some users post their public key on their website.

... But websites can be hacked!

Problem:

Person X wishes to send an encrpyted message to Y.
How can X be sure that he has Y's real public key?
If he uses the wrong key, the message contents could be decoded!

Solution:

"Digitial Certificates"

Digital Certificates

Assume that there is a trusted third party.

"The Authenticator"

Everyone knows the true public key of the authenticator.

Y registers with the authenticator first.

After verifying Y's identity (e.g., checking driver's license) the authenticator issues Y a ***"certificate."***

The certificate contains:

- **Y's identity (name, website URL, email, etc.)**
- **Y's public key**

...and is digitally signed by the authenticator!

Digital Certificates

Assume that there is a trusted third party.

"The Authenticator"

Everyone knows the true public key of the authenticator.

Y registers with the authenticator first.

After verifying Y's identity (e.g., checking driver's license) the authenticator issues Y a ***"certificate."***

The certificate contains:

Y's identity (name, website URL, email, etc.)

Y's public key

...and is digitally signed by the authenticator!

Y begins by sending X his certificate.

X uses the authenticator's public key to learn Y's public key.

X then uses this public key to send Y a secure message.

User Authentication: Passwords

Approach #1:

OS maintains a file of <username, password> pairs.

User Authentication: Passwords

Approach #1:

OS maintains a file of <username, password> pairs.

Password cracking?

User Authentication: Passwords

Approach #1:

OS maintains a file of <username, password> pairs.

Password cracking?

Build a list of...

- **Use person's name**
- **Use telephone book of all last names**
- **Use list of first names**
- **Common city and street names**
- **Short random strings ("xyz", "12345")**
- **Sexual terms, obscene expressions**
- **All of the above spelled backwards**

User Authentication: Passwords

Approach #1:

OS maintains a file of <username, password> pairs.

Password cracking?

Build a list of...

- **Use person's name**
- **Use telephone book of all last names**
- **Use list of first names**
- **Common city and street names**
- **Short random strings ("xyz", "12345")**
- **Sexual terms, obscene expressions**
- **All of the above spelled backwards**

Over 86% of passwords appeared on this list.

Try them one-by-one, automatically!

User Authentication: Passwords

Try them one-by-one, automatically!

A ***"war dialer"***

- **A dedicated computer...**
  - **Using a modem to call random numbers**
  - **If a computer answers, start trying passwords!**

Over the internet...

- **Every computer has a 32-bit IP address**
  - **123.123.123.123**
  - **telnet 123.123.123.123**

User Authentication: Passwords

Try them one-by-one, automatically!

A *"war dialer"*
- **A dedicated computer...**
  - **Using a modem to call random numbers**
  - **If a computer answers, start trying passwords!**

Over the internet...
- **Every computer has a 32-bit IP address**
  - **123.123.123.123**
  - **telnet 123.123.123.123**

Solution: Slow the cracker down
- **Break the telnet connection after 3 unsuccessful attempts.**

User Authentication: Passwords

Try them one-by-one, automatically!

A *"war dialer"*
- **A dedicated computer...**
  - **Using a modem to call random numbers**
  - **If a computer answers, start trying passwords!**

Over the internet...
- **Every computer has a 32-bit IP address**
  - **123.123.123.123**
  - **telnet 123.123.123.123**

Solution: Slow the cracker down
- **Break the telnet connection after 3 unsuccessful attempts.**

Counter-threat:
- **Attack many computers in parallel!**

User Authentication: Passwords

Generate random IP addresses
Look for any computer
Try to break in

"Script Kiddies"
- **Technically naïve users**
- **Run scripts they find on the internet**

New computers will be attacked...
- **as soon as they are connected to internet!**

User Authentication: Passwords

How a "login program" works:

- **Read in username and password**
- **Run the password through a one-way encrpytion function**
- **Read the password file**
- **Compare the encrypted password to what's in the file**

  (Password file does not store the passwords directly!)

User Authentication: Passwords

How a "login program" works:

- **Read in username and password**
- **Run the password through a one-way encrpytion function**
- **Read the password file**
- **Compare the encrypted password to what's in the file**

  (Password file does not store the passwords directly!)

Method of Attack:

- **Build a list of likely passwords**
- **Use the one-way encryption function to encrypt each**
- **Build a list of < password, E(password) > pairs**
- **Read the password file**
- **Look for encrpyted passwords that match something in this list**
- **Get the corresponding password from the list**

User Authentication: Passwords

"Salting the Password File"

Each user is assigned a random number (the "salt").

Each time the password is changed,
the user is assigned a new number.

The password is concatenated with the salt.

The *combination* is encrpyted:

`E(password||salt)`

The password file contains a list of triples:

`< username, salt, E(password||salt) >`

The login program...

- Reads the password file to get the salt
- Computes `E(password||salt)`
- Compares it to what is stored in the password file

The cracker cannot pre-compute encrypted passwords!

User Authentication: Passwords

Restricting access to the password file:

Make the password file unreadable
... except by a special system routine

If you want to read the password file...
Must call a system routine to do the reading
... specifying which username you want
The routine introduces a delay
... which makes it difficult to try lots of testing!

User Authentication: Passwords

Must educate (or force) users to choose "good" passwords!

Choosing a password:

- **Minimum of 7 characters**
- **Contain upper and lower case letters**
- **Contain digits and special symbols**
- **Not contain dictionary words, names, etc.**
- **Don't use same password as for another (insecure?) system**

User Authentication: Passwords

Must educate (or force) users to choose "good" passwords!

Choosing a password:

- **Minimum of 7 characters**
- **Contain upper and lower case letters**
- **Contain digits and special symbols**
- **Not contain dictionary words, names, etc.**
- **Don't use same password as for another (insecure?) system**

But if passwords are hard to remember...

users will write them down

...which may compromise security!

Problem: Packet Sniffers

Most communication is insecure!
What if someone is watching the communication?
They see what password (or encrpyted password)
you send over the Internet
Later, they supply the same data
How can the server protect against this?

Problem: Packet Sniffers

Most communication is insecure!
What if someone is watching the communication?
They see what password (or encrpyted password) you send over the Internet
Later, they supply the same data
How can the server protect against this?

One-Time Passwords

Issue each user a list of passwords.
Each one can only be used once.

Reasonable in some situations...
... but not very convenient for most of us.

Problem: Packet Sniffers

Leslie Lamport

Assume we have a one-way encryption function, f

User chooses a normal password, P.
From this, we can generate N one-time passwords.

The 1st password is $f(f(f(f(P))))$
The 2nd password is $f(f(f(P)))$
The 3rd password is $f(f(P))$
The 4th password is $f(P)$

(This example uses 4, but often a much large number)

Problem: Packet Sniffers

Leslie Lamport

Assume we have a one-way encryption function, *f*

User chooses a normal password, P.
From this, we can generate N one-time passwords.

The 1st password is $f(f(f(f(P))))$
The 2nd password is $f(f(f(P)))$
The 3rd password is $f(f(P))$
The 4th password is $f(P)$

(This example uses 4, but often a much large number)

If a cracker sees one password...

It is easy to compute the previous password.
It is impossible to compute the next password!

Problem: Packet Sniffers

Leslie Lamport

Assume we have a one-way encryption function, *f*

User chooses a normal password, P.
From this, we can generate N one-time passwords.

- **The 1st password is** $f(f(f(f(P))))$
- **The 2nd password is** $f(f(f(P)))$
- **The 3rd password is** $f(f(P))$
- **The 4th password is** $f(P)$

(This example uses 4, but often a much large number)

The server stores...

$f(P)$ and i, which indicates which password to use next.

During login, the client asks the server for the value of i.
The client asks user for P and computes i-th password.
After login succeeds, the server increments i.

Challenge-Response Authentication

This system employes a one-way function, *f*(plaintext,key).
Each user (client) chooses a secret key, K.
Server will also know the user's key.

Upon login, the server will generate a random number, R.
Server sends R to the user's machine.
The user computes *f*(R,K) and sends it to server.
The server also computes *f*(R,K) and compares it.

An eavesdropper will see R and *f*(R,K).
Cannot deduce the secret key, K.

Authentication with Cards

Magnetic Stripe Cards

140 bytes of info

Contain username and encrypted password

Cost: $0.10 to $0.50

Chip-Based Cards

Stored-Value Cards

EEPROM with ~ 1Kbyte of info

No CPU

Cost: $1.00

Can store, e.g., encrypted info about phone credit

Smart Cards

8Mhz CPU, 16Kbyte ROM, 4Kbyte EEPROM, Scratch RAM, 9600 baud communication channel

Cost: $5.00 to $50.00

Authentication with Cards

Smart Cards - Usage...

Encode complex info about user

Personal health records

Card can engage in complex authentication protocols

Problem:

The point-of-use machine may be insecure.

The card can communicate directly with the remote server!

(The point-of-use machine is no more than an insecure communication channel.)

If the cryptographic protocol is broken in the future, the card can be re-programmed.

Even now, some cards contain a Java interpreter!

Authentication with Biometrics

Enrollment

The user's characteristics are measured

Stored on server / smart card

Identification

The user's characteristics are measured and compared

The match is never exact...

False negatives - User is annoyed at being forbidden entry.

False positives - How many can be tolerated?

Authentication with Biometrics

Fingerprint

Finger-Length Measurement

Can be attacked with a mold or *... (yuk) ...*

Retinal Pattern Analysis

Blood vessels vary, like fingerprints, even in twins

Can be accurately photographed from 1 meter!

Storage required: ~256 bytes

Spoofing? Also look for the pulsing heartbeat in the vessels.

Signature Analysis

User signs with a pen which measures position, velocity, pressure

Voice Biometrics

Easier than voice *recognition*.

Spoofing? Must use challenge-response!

Useful over the telephone.

Authentication - Countermeasures

Call-Back

Upon successful login...

Server breaks connection and calls the user back.

Server then knows your phone number.

Remind User of Last Login

Upon successful login...

Server shows the date and time of last login

The user can detect if his account was misused

Laying Bait / Sting Operations

Intentionally create security loopholes

Monitor them and, when attacked, catch the culprit

Trojan Horses

A program that looks okay, but contains malicious code!

The bad-guy distributes a program that seems desirable.
When run by an unsuspecting user...
- **It may do what it is supposed to do.**
- **But it may also do bad things on your machine!**

Trojan Horses

A program that looks okay, but contains malicious code!

The bad-guy distributes a program that seems desirable.

When run by an unsuspecting user...

- It may do what it is supposed to do.
- But it may also do bad things on your machine!

The program runs on your computer with no protection.

- Can read / update / erase all your files
- Can send your private info over the Internet
- Can create trapdoors for later use:
  - The bad guys can log directly on to your computer to look around for anything interesting.

Trojan Horses

A program that looks okay, but contains malicious code!

The bad-guy distributes a program that seems desirable.

When run by an unsuspecting user...

- It may do what it is supposed to do.
- But it may also do bad things on your machine!

The program runs on your computer with no protection.

- Can read / update / erase all your files
- Can send your private info over the Internet
- Can create trapdoors for later use:
  - The bad guys can log directly on to your computer to look around for anything interesting.

You may never know that anything bad happened!

Trojan Horses

A program that looks okay, but contains malicious code!

The bad-guy distributes a program that seems desirable.
When run by an unsuspecting user...
- **It may do what it is supposed to do.**
- **But it may also do bad things on your machine!**

The program runs on your computer with no protection.
- **Can read / update / erase all your files**
- **Can send your private info over the Internet**
- **Can create trapdoors for later use:**
  - **The bad guys can log directly on to your computer to look around for anything interesting.**

You may never know that anything bad happened!

Downloading programs from Internet is getting more common!
Have you ever downloaded a program and then run it?

Trojan Horses on Unix

Goal: get another user (e.g., super-user) to execute a program you wrote (without knowing it).

UID - User ID

- Determines the privileges of a running program.
- Normally, a program has privileges of the user running it.
- Example: When you execute the “shell”” program.
- UID is carefully controlled by the OS.

When you type a command...

- $PATH tells where to find a program with that name.
- Typically, the system will look in
  - the current directory “.”
  - /bin
  - /usr/bin
  - *...etc...*

Trojan Horses on Unix

Idea:

Create a program called "ls"
- **A trojan horse, in your directory**
- **Will (seem to) work just like "ls"**

Ask super-user for help.
- **The super-user logs in (as root) and types:**

```
cd <your directory>
ls
```

The super-user executes the trojan horse with root privileges!

Your trojan horse "ls" will also...
- **`chown` of your *personal* shell to "root".**
- **`chmod` of your shell to set its SETUID bit.**

Now, whenever you run your new shell...
- **You can do anything super-user can!**

Trojan Horses on Personal Computers

The Trojan Horse is run on someone’s PC...

The program...
- Looks to see if you use “Quicken”
  - A personal accounting / online-banking program
- Asks the program to transfer money
  - ...from your account, to an account in a foreign country

Trojan Horses on Personal Computers

The Trojan Horse is run on someone's PC...

The program...

- **Looks to see if you use "Quicken"**
  - **A personal accounting / online-banking program**
- **Asks the program to transfer money**
  - **...from your account, to an account in a foreign country**

This worked!

Trojan Horses on Personal Computers

The Trojan Horse is run on someone's PC...

The program...

- **Looks to see if the computer is attached to a dial-up modem.**
- **Instructs the modem to dial a "900" number**
  - **...in Moldova (charging $10/minute to use)**
- **When the user is using the Internet...**
  - **the internet provider is now in Moldova**
  - **the user doesn't notice anything, stays on for hours.**
- **The user gets a huge phone bill.**
- **The local company is supposed to re-imburse the Internet provider in Moldova**

Trojan Horses on Personal Computers

The Trojan Horse is run on someone's PC...

The program...

Looks to see if the computer is attached to a dial-up modem.
Instructs the modem to dial a "900" number
...in Moldova (charging $10/minute to use)
When the user is using the Internet...
the internet provider is now in Moldova
the user doesn't notice anything, stays on for hours.
The user gets a huge phone bill.
The local company is supposed to re-imburse the Internet provider in Moldova

This worked!

800,000 minutes, 38,000 victims, $2.74 million in charges

Login Spoofing

On a multi-user system...

- **Users sit at public terminals and login.**
- **The “login” program asks for a username and password.**

The Idea:

- **The bad guy runs a program on one of the terminals.**
- **This program asks for a username and password.**
  - **Looks just like the “login” program**
- **Innocent user comes by & types in username and password**
- **The program...**
  - **Saves password.**
  - **Displays: “login incorrect, try again”**
  - **Terminates and quitely logs out.**
  - **The true “login” program starts.**
- **The user doesn’t know he was duped.**

Logic Bombs

A programmer writes software for his employer...

When companies fire / layoff an employee

Guards will escort him to the door

... To prevent theft / damage of information

Idea:

The programmer secretly inserts malicious code into the company's software

Example:

The code requires a new password every day.

If not supplied...

the program does damage to company files.

Encrypts them / deletes them.

If the employee is terminated...

He gets revenge when all their files are deleted!

Or, he can use blackmail with the decryption key!

Logic Bombs

Counter-measure: Code Walk-Throughs

All members of the programming team get together.
The programmer walks them through the code he wrote.
He explains how it works.

Also good for catching bugs!

Trap Doors

```
repeat
  Print ("Enter login:")
  GetString (username)
  TurnOffEchoing ()
  Print ("Enter password:")
  GetString (password)
  TurnOnEchoing ()
  v = CheckPassword (username, password)



until v == true
```

Trap Doors

```
repeat
  Print (“Enter login:”)
  GetString (username)
  TurnOffEchoing ()
  Print (“Enter password:”)
  GetString (password)
  TurnOnEchoing ()
  v = CheckPassword (username, password)
  if password == “xyzzy”
    break
  endIf
until v == true
```

Buffer Overflow

Array bounds checking is not done in "C".
The source of many subtle bugs!

```
char [100] myBuffer;
...
i = 100000;
myBuffer[i] = 'x';
```

Some byte outside of the array is altered!

Buffer Overflow

Array bounds checking is not done in "C".
The source of many subtle bugs!

```
char [100] myBuffer;
...
i = 100000;
myBuffer[i] = 'x';
```

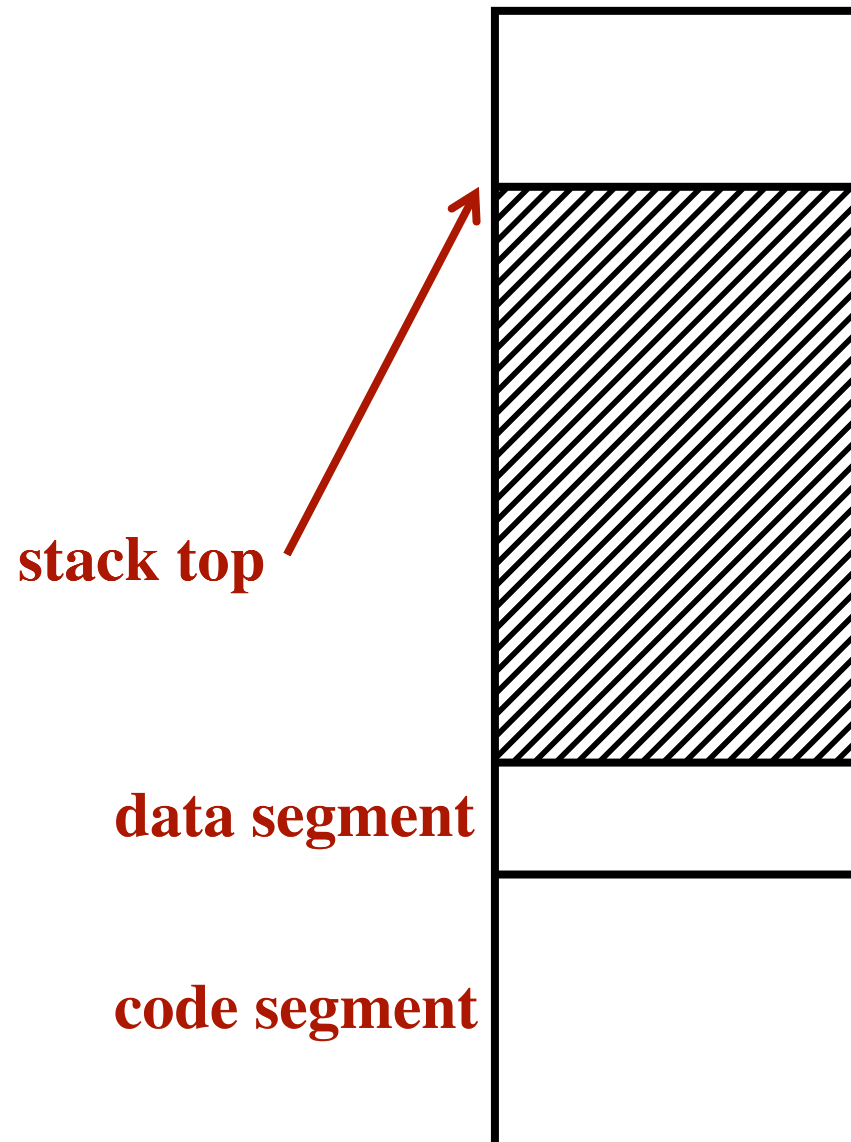
Some byte outside of the array is altered!

Consequences:

- **Program still works correctly on all inputs**
- **Program produces erroroneous output**
  - **Obvious, noticed**
  - **Not obvious, not noticed as an error!**
- **Program has "*erratic*, *undefined*" behavior**
  - **Program crashes**
  - **Program has other unexpected behavior**

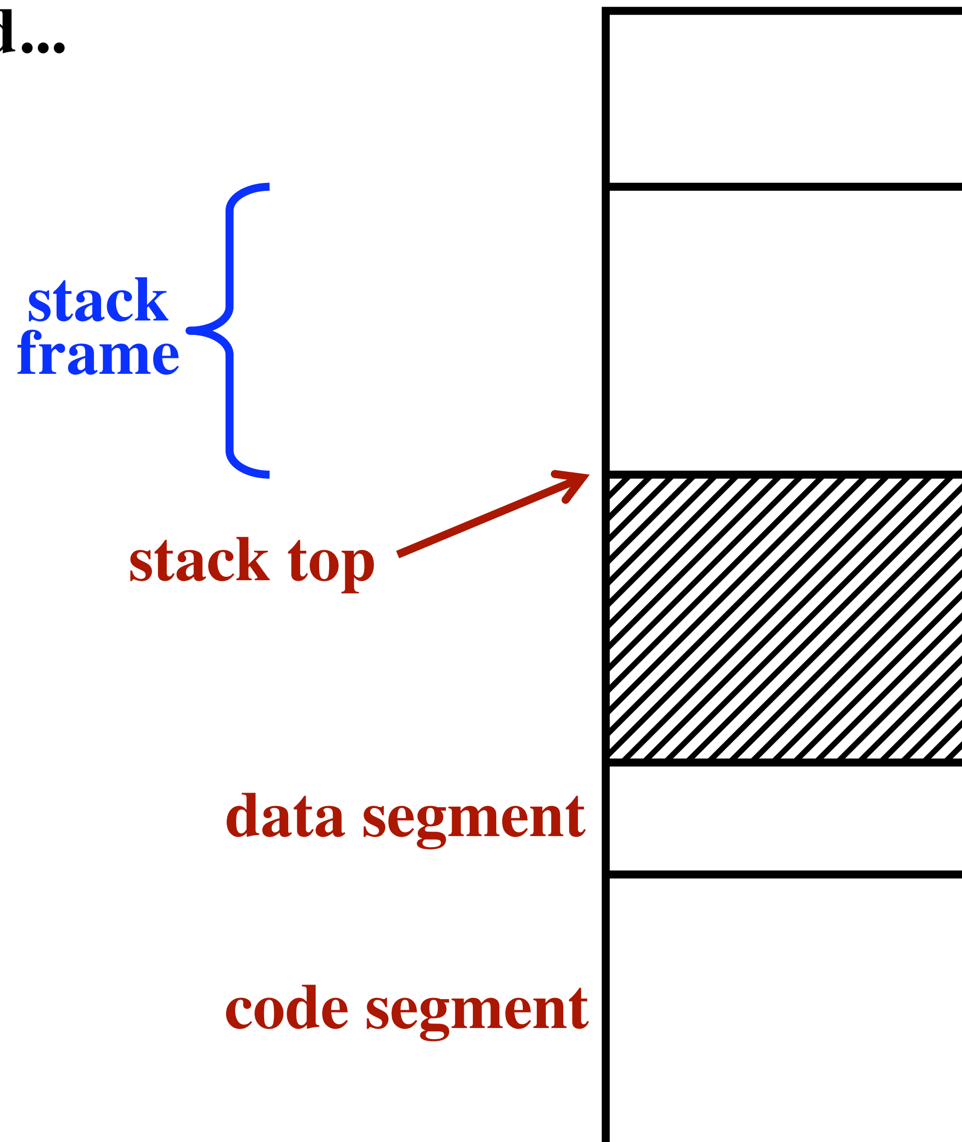
Buffer Overflow

A subroutine is called...



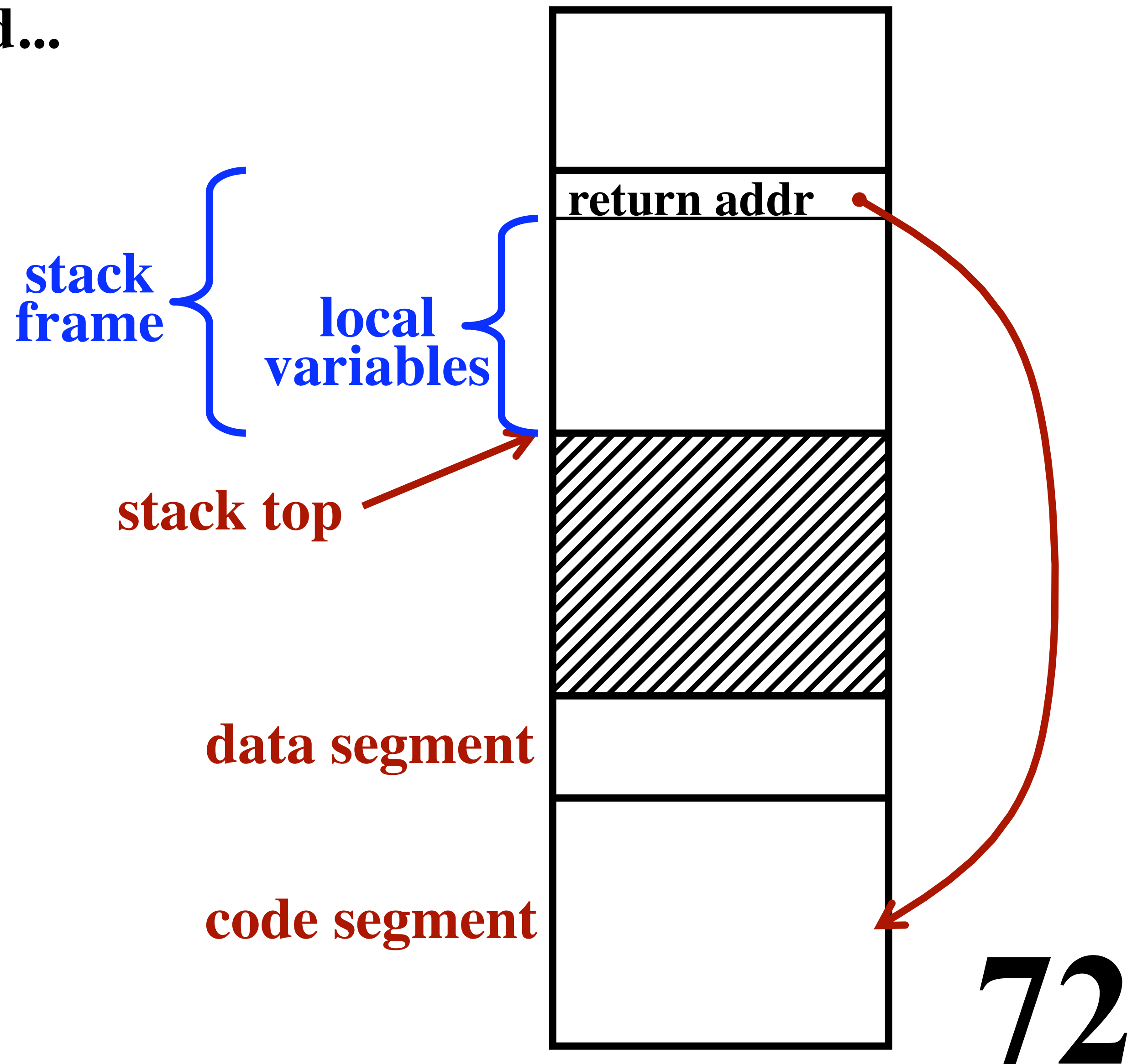
Buffer Overflow

A subroutine is called...



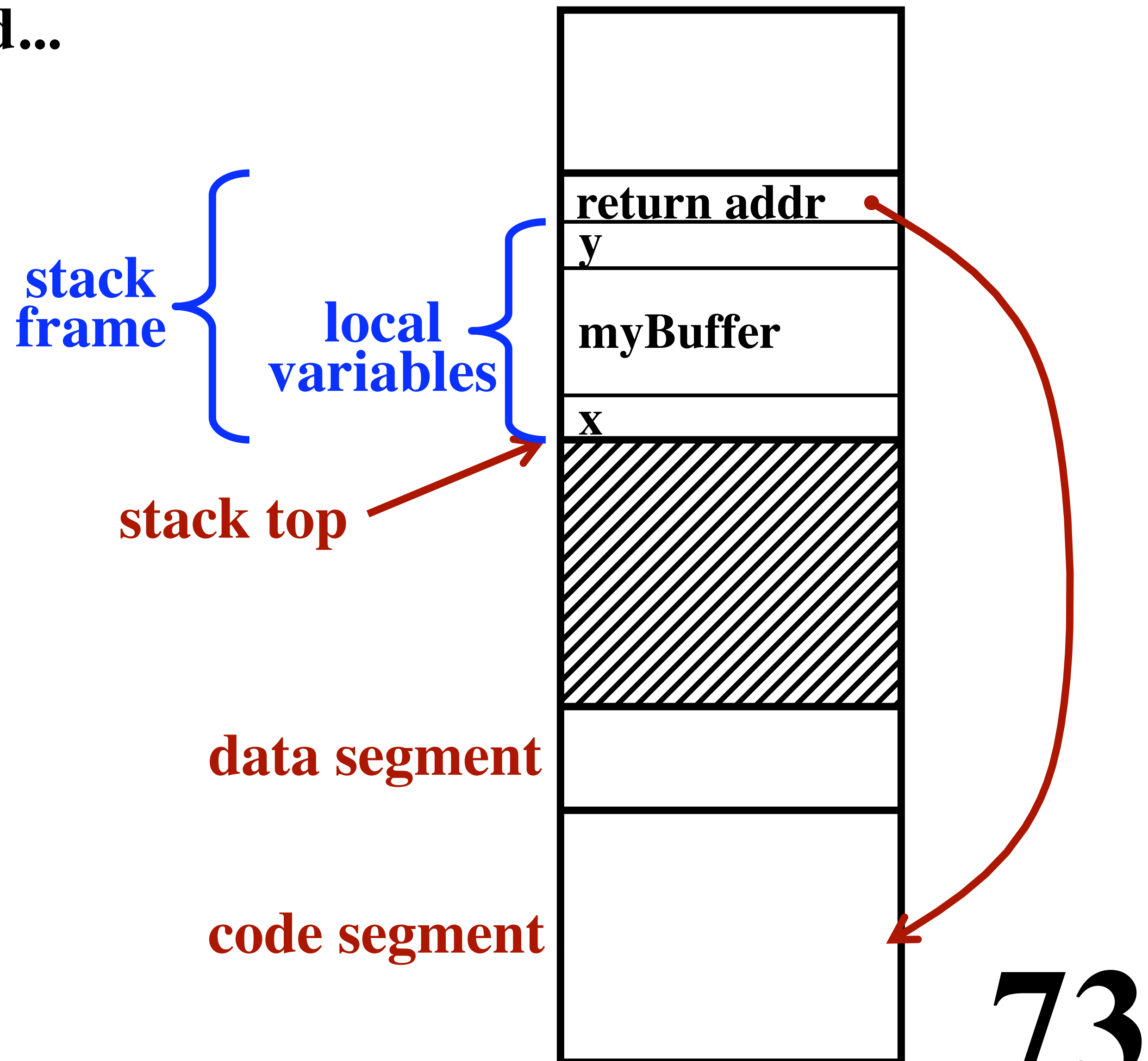
Buffer Overflow

A subroutine is called...



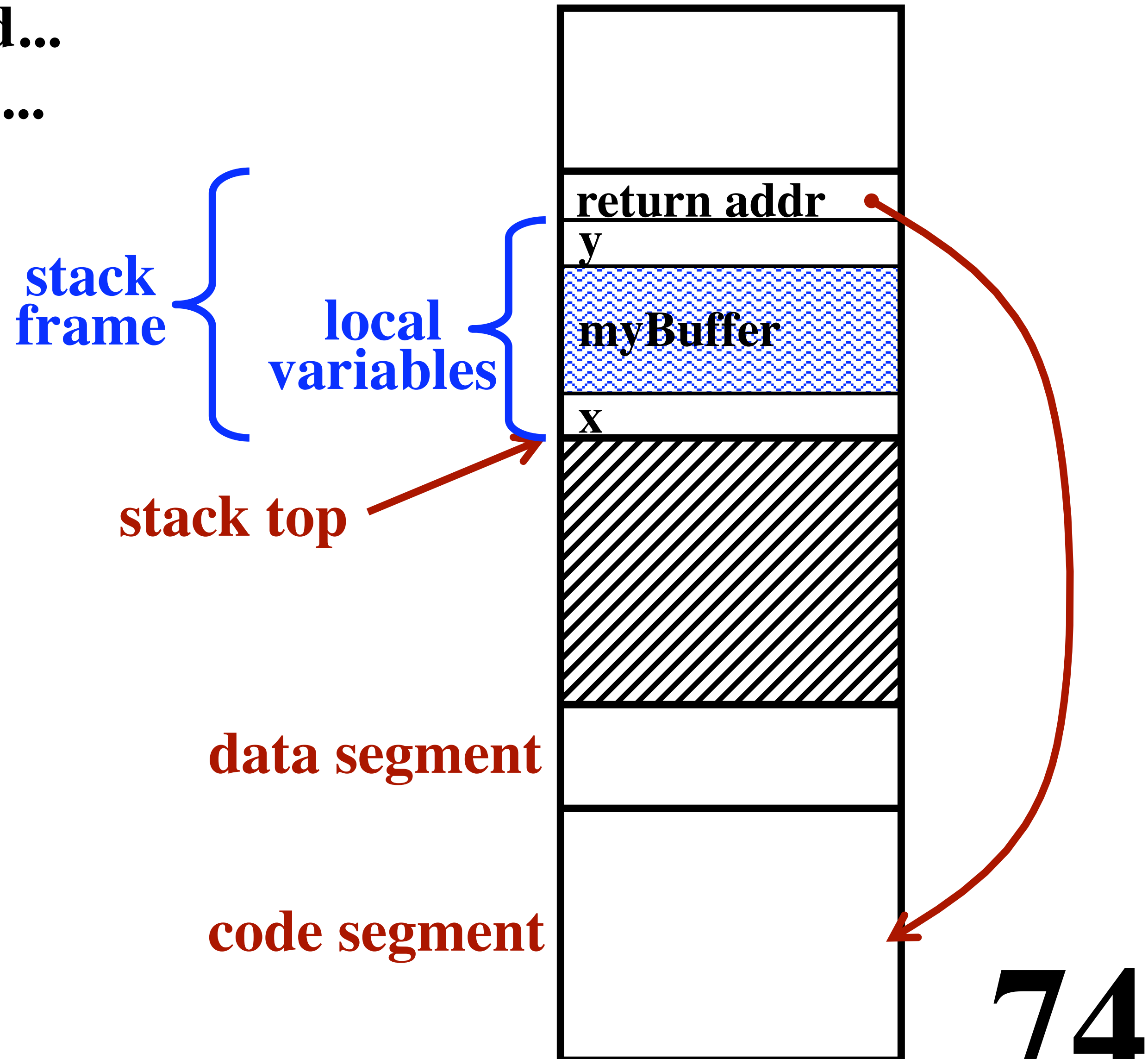
Buffer Overflow

A subroutine is called...



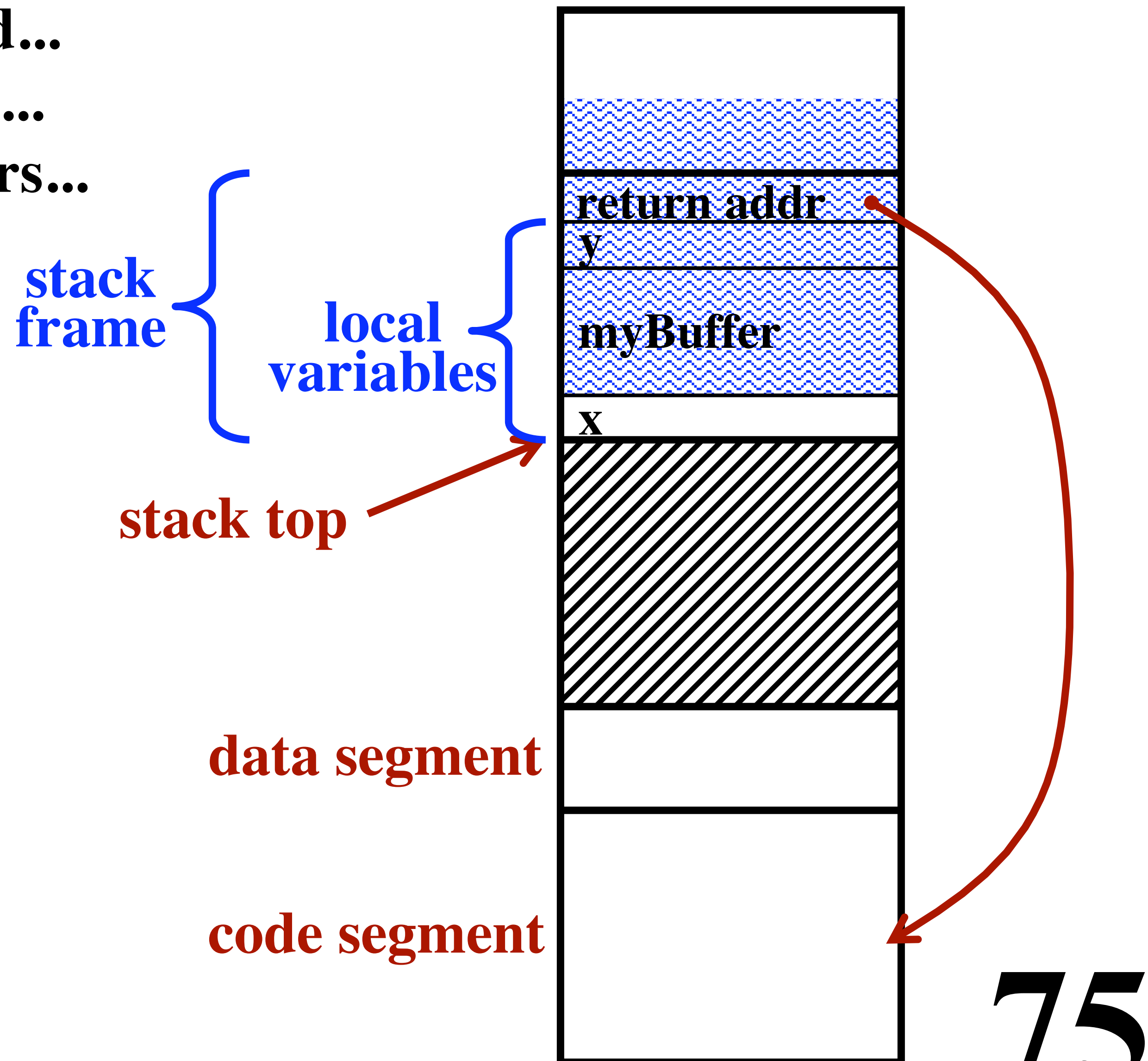
Buffer Overflow

A subroutine is called...
The array is updated...



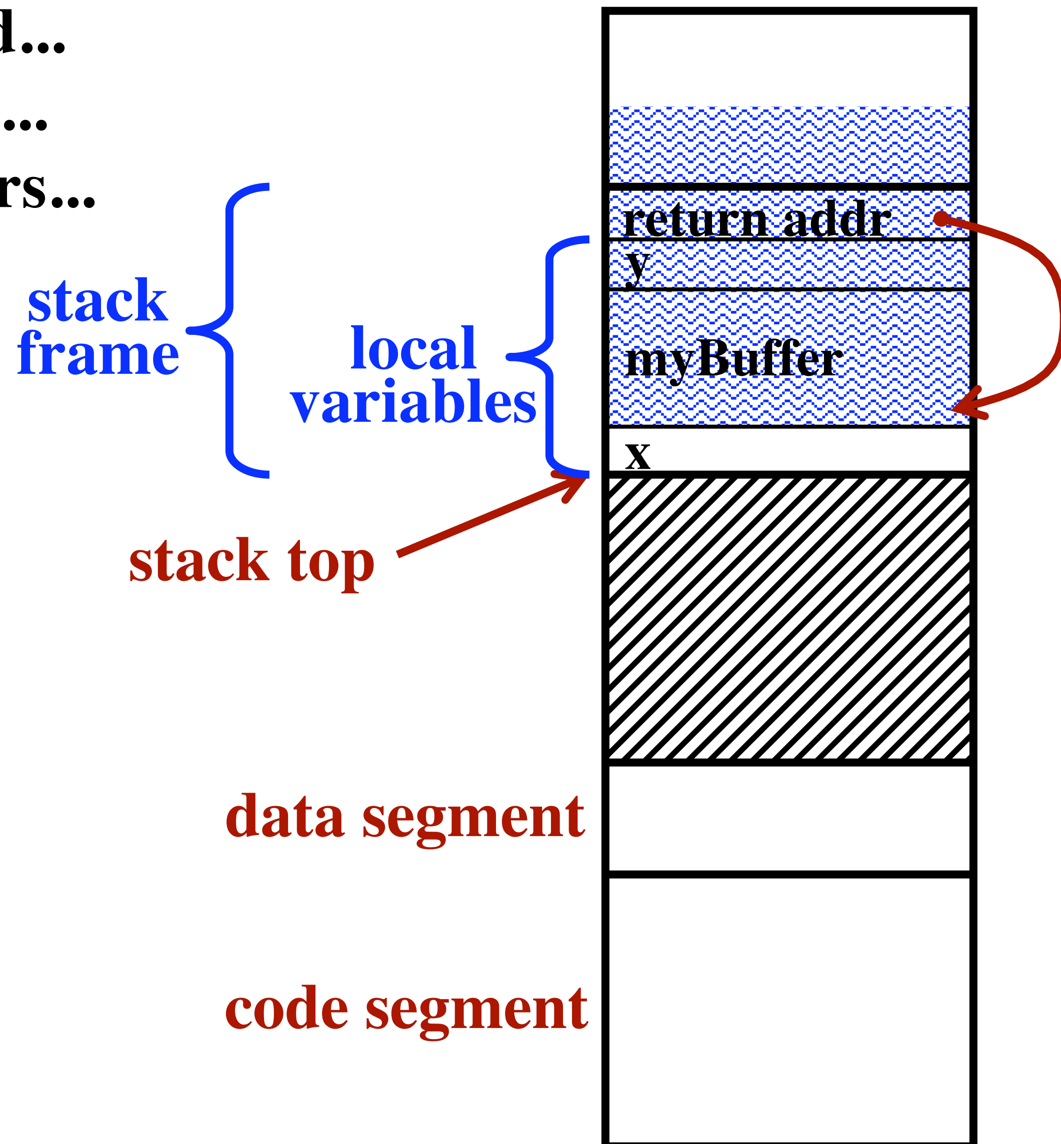
Buffer Overflow

A subroutine is called...
The array is updated...
Buffer overflow occurs...



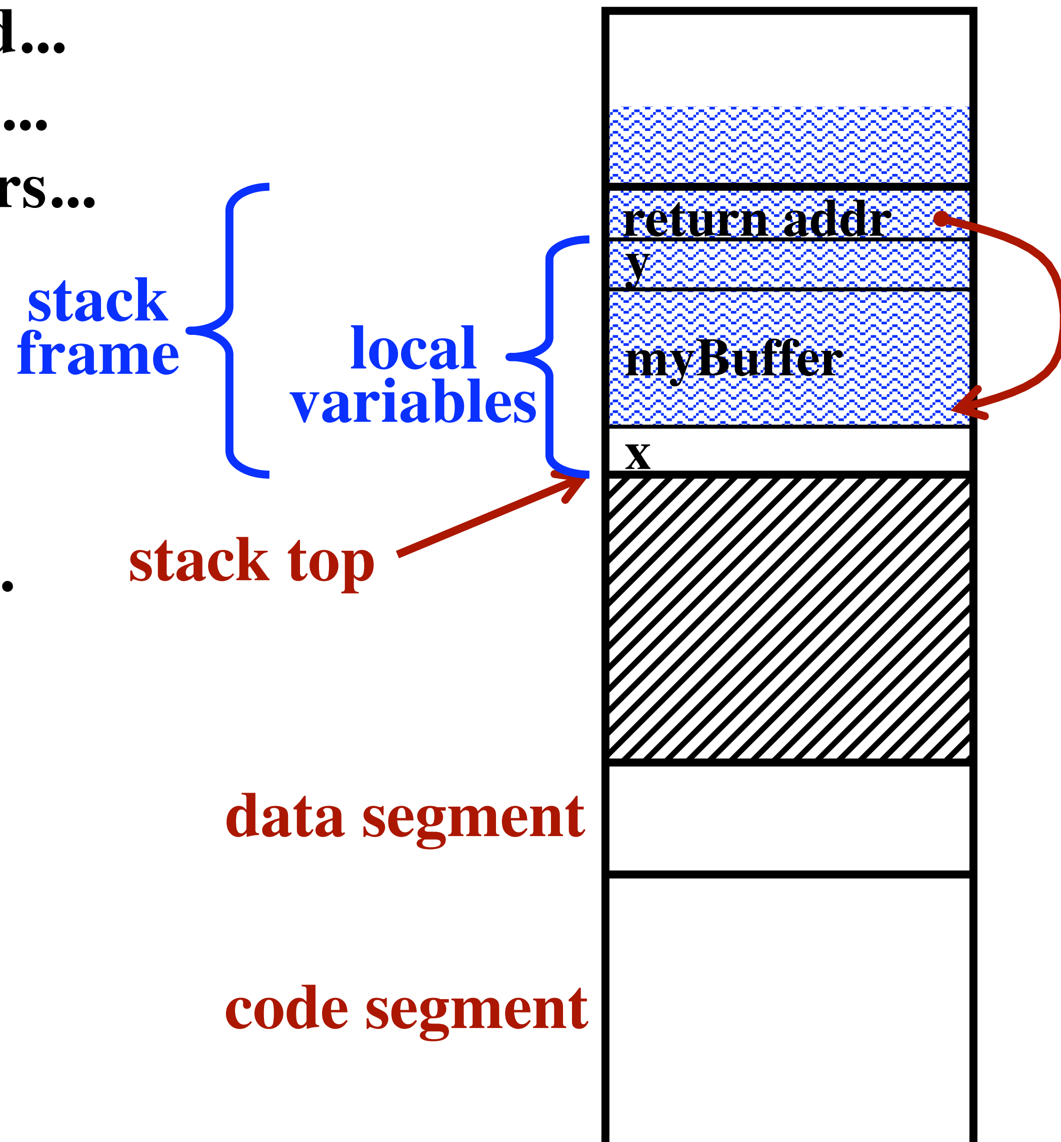
Buffer Overflow

A subroutine is called...
The array is updated...
Buffer overflow occurs...
The return address
is overwritten
with a carefully
planned value!



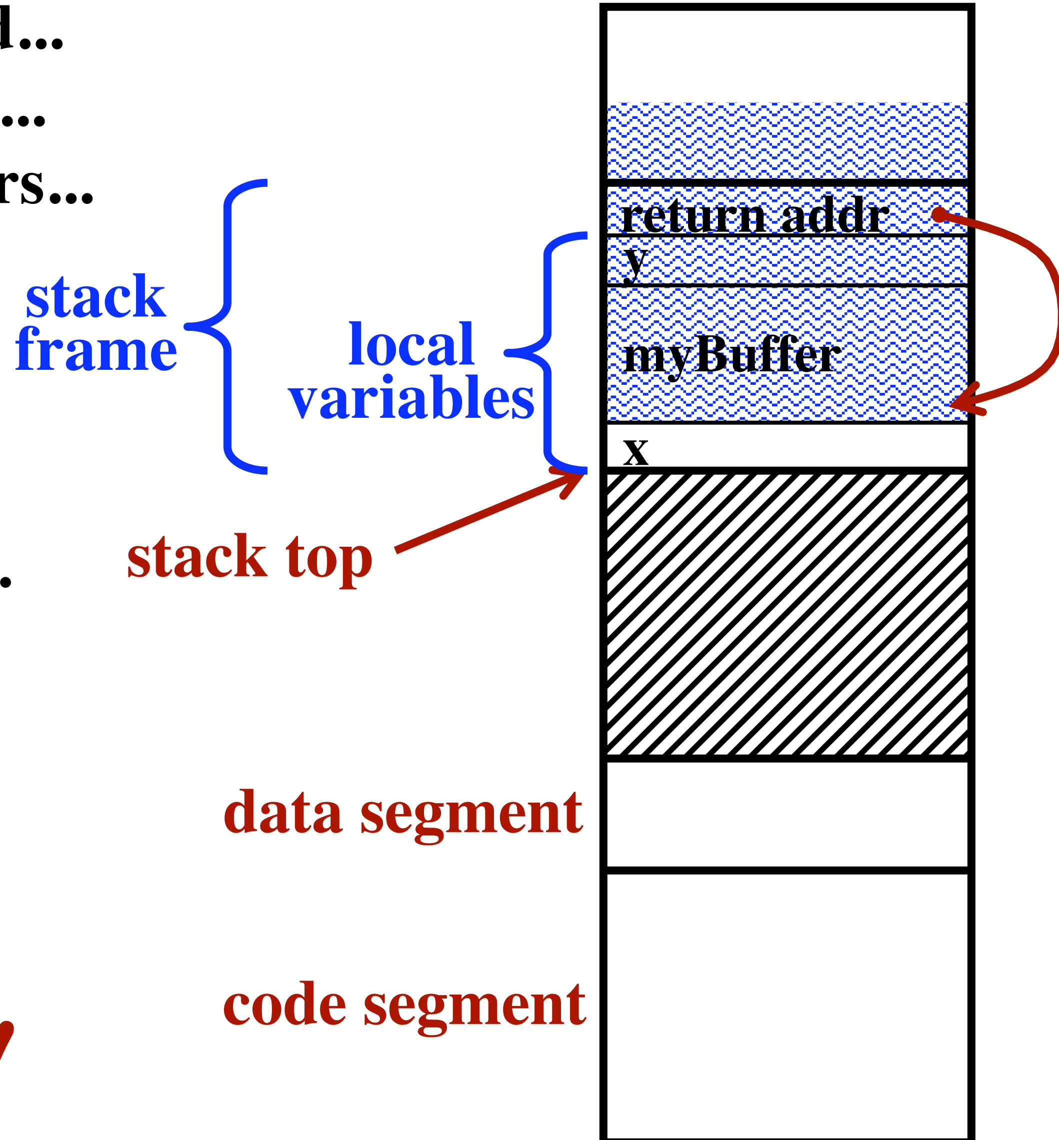
Buffer Overflow

A subroutine is called...
The array is updated...
Buffer overflow occurs...
The return address
  is overwritten
  with a carefully
  planned value!
The routine returns...



Buffer Overflow

A subroutine is called...
The array is updated...
Buffer overflow occurs...
The return address
is overwritten
with a carefully
planned value!
The routine returns...



Arbitrary Code Can Be Executed!!!

A Famous Security Flaw

Tenex / DEC-10

User program could request notification whenever it had a page fault.

The user program provided passwords in user-space.

The kernel would check passwords, one character at a time.

Upon a mismatch, it would stop *immediately* and report failure.

Method of attack:

- **User program carefully aligns the password on a page boundary.**
- **First page is in memory, second page is not.**
- **The first character lies on 1st page; remainer on 2nd page.**
- **If the kernel reports a page fault, first character was OK.**
- **If the kernel reports password failure, first char was wrong.**

256 tries to guess the learn the first char of password.

8 x 256 tries to learn an 8 character password, not 256^8.

Design Principles for Security

[Quotes from Tanenbaum]

"The system design should be public."

The bad guys will eventually discover design details.

If secrecy of design is critical, then once the bad guys find out the design, security is lost forever.

It is a delusion for designers to seek *"security from obscurity"*.

"The default should be 'no access'."

Any errors in the protection mechanism will get reported.

However, if the default is 'access allowed', any flaws in the protection mechanism will go unreported!

"Check for current authority."

Example: User opens a file and keeps it open for weeks.

Often, an OS checks permissions during the `open` syscall.

The permissions could change while a file remains open.

When a `read` occurs, does the system check again?

Design Principles for Security

[Quotes from Tanenbaum]

"Give each process the least privilege possible"

Use a fine-grained security mechanism.
We want to limit the damage a Trojan Horse can do.

"Protection mechanism should be simple, uniform, and built into the lowest layers of the system."

Security cannot be retrofitted onto an insecure system.
"Security, like correctness, is not an add-on feature."

"The security scheme must be psychologically acceptable."

Or else users will not use it.

"Keep the design simple."

"A system with many features is a big system...
The more code there is, the more security holes and bugs there will be."

Viruses

Code is transferred from a remote computer (over network) and executed on the target machine.

Most virus writers seem to be students or recent students. They seem to approach it as a technical challenge.

Virus:

Code that attaches itself to another program.
It can reproduce itself & spread the infection.
It may also cause harm.

Worm:

Similar to a virus: It can self-replicate.
Independent programs, not "attached" to other programs.
(Not an important distinction.)

Viruses

What harm can they do?

Anything a legitimate program can do!

- **Harmless stuff, e.g., display a message on the screen.**
- **Delete or subtlely corrupt files.**
- **Steal information, e.g., send your data over Internet.**
- **Encrypt your files, for blackmail purposes.**
- **Make the computer unusable**
  - **"denial of service" attack**
  - **Use up CPU cycles, fill up the disk, etc.**
- **Update the BIOS in FLASH memory.**
  - **Computer will no longer boot properly.**
  - **Fixing this may involve a service call.**
- **The virus may only target a specific group of computers.**
  - **E.g., If the computer is ever used to browse an al-Qa'ida website, then erase all files.**

How Viruses Work

Usually written in assembly code.
A file is intentionally infected.
The file is put "out there" to be shared.

- **Open / Free software**
- **Pirated version of some well-known program**

When the infected program is executed...

Step 1: **Infect some other files.**
Step 2: **Execute its "*payload*".**

The Payload

E.g., delete all files.
May do nothing at all.
May wait until a specific date to execute the payload.
Allows it to spread without people noticing it.
Then, once widespread, it can do more harm.

Companion Viruses

The users thinks he is running one program.
Another program (the companion) is executed instead.
The companion ends by jumping to the desired program.

Example:

On MS-DOS...

User types a command "`prog`".
A file called "`prog.exe`" is executed.

Method of attack:

The OS looks for "`prog.com`" first, then "`prog.exe`".
Most programs are "`.exe`".
The companion virus is called "`prog.com`".

First: It does its dirty work.
Second: It executes "`prog.exe`".

Virus Code

Algorithm:

Save the virus's code in an array in memory.
Walk the entire directory structure.
For every file you find...
  Open the file
  Overwrite the file with the virus's code.
Use a recursive tree walking algorithm
  to visit all sub-directories.
In order for newly inflected files to be executable,
  they must be in "executable" (a.out) format.
Idea: Initialize the array by reading
  from the initial executable file
    (E.g., open file argv[0] and read from it.)

Virus Code

Problem:

- Successful (biological) infections must not kill too quickly.
- The infected host must be alive so it can spread the infection.

Improvement:

- Only infect 1% of the files.
- Use a random number generator to decide whether or not to infect a file.

Improvement:

- Regulate the rate of infection.
- Only infect 0.1% of files per day.

To avoid detection:

- Try to avoid updating time-of-last-modification.
- Do not change the file size.

Anti-Virus Code

Consider the algorithm used by the virus to spread...

An anti-virus program could use the exact same logic.

The anti-virus program would visit the exact same files.
Look for infections only where they would occur.

Virus Code

Problem with overwriting viruses:

The original program is rendered useless.

The virus is detected by the user immediately.

Goal: **Hide by letting the original program function okay.**

"Parasitic Viruses"

Virus code is *added* to the infected program.

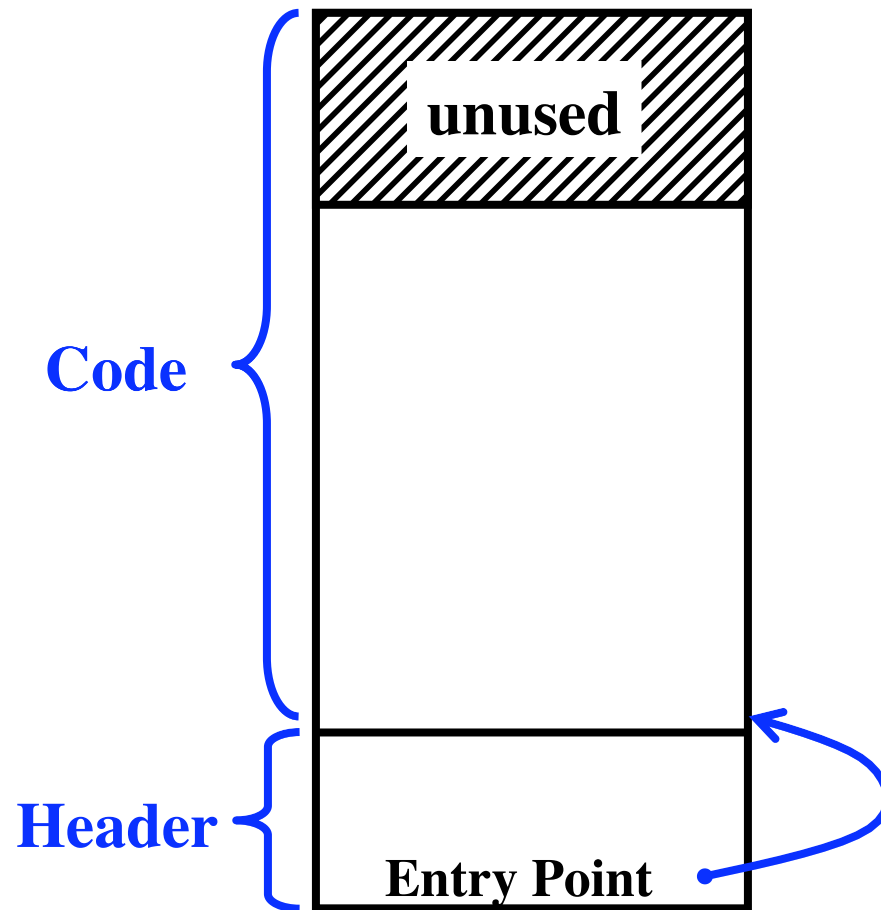
The infected program continues to function correctly.

The virus code must be "squeezed" into the program.

"Cavity Viruses"

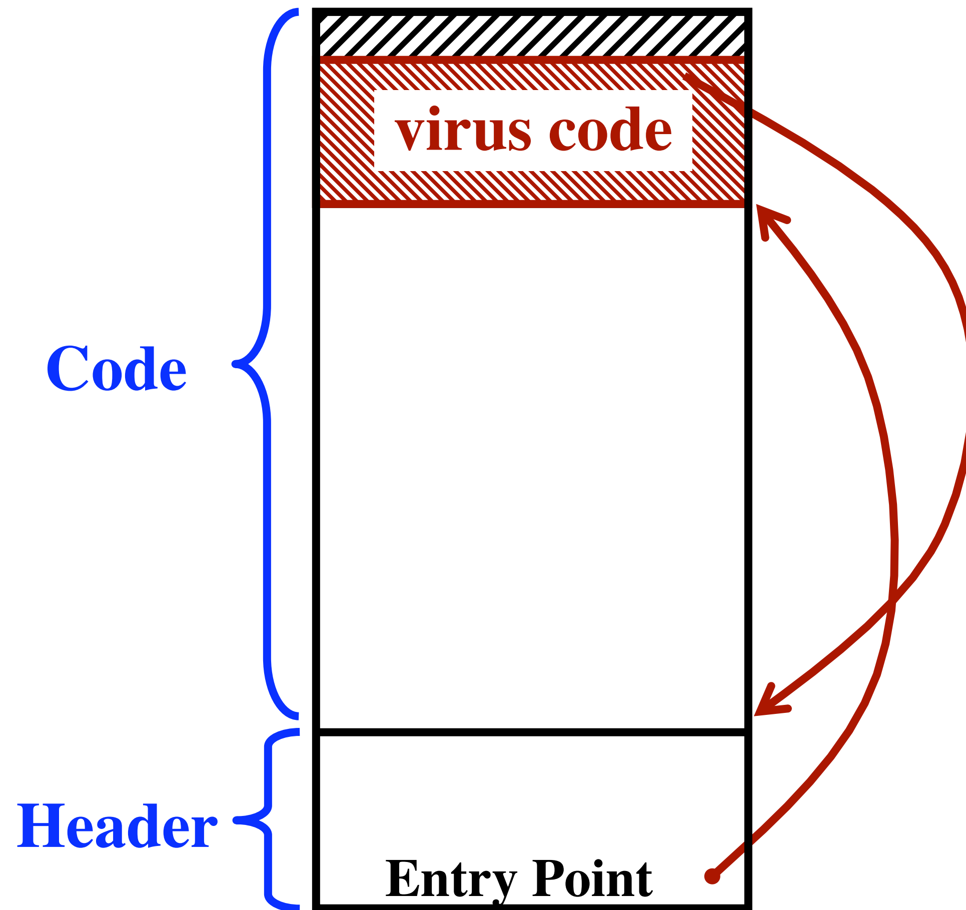
Utilize extra, unused space in the executable file.

Cavity Viruses



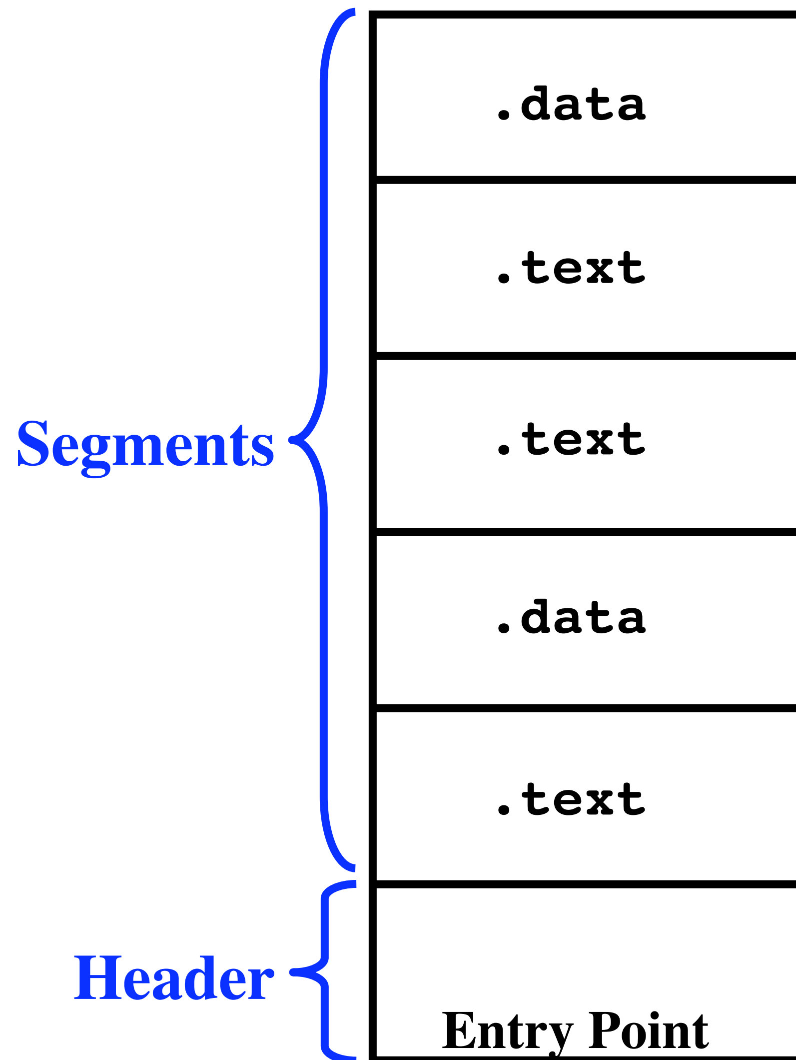
Executable File (a.out)

Cavity Viruses

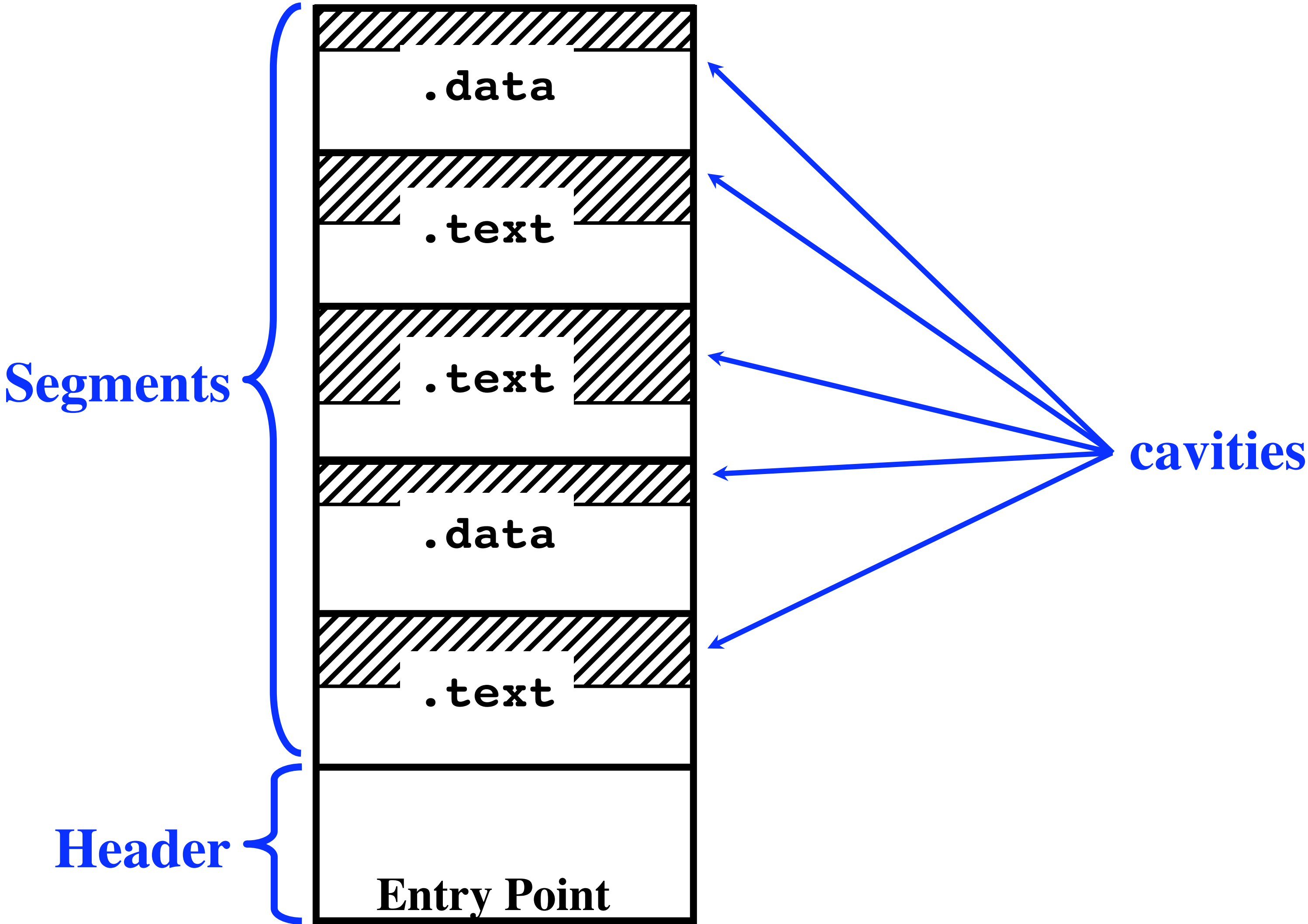


Executable File (a.out)

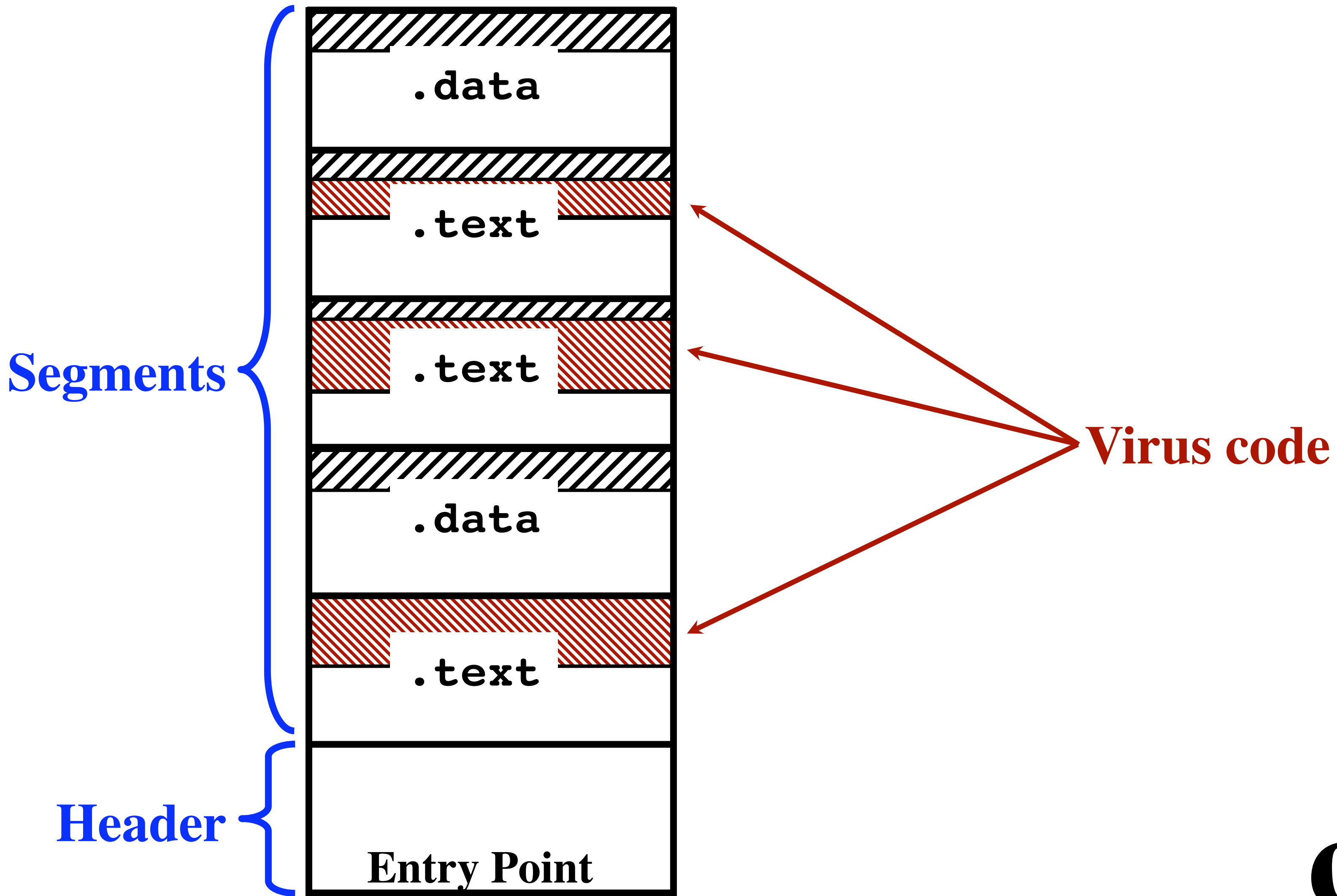
Cavity Viruses



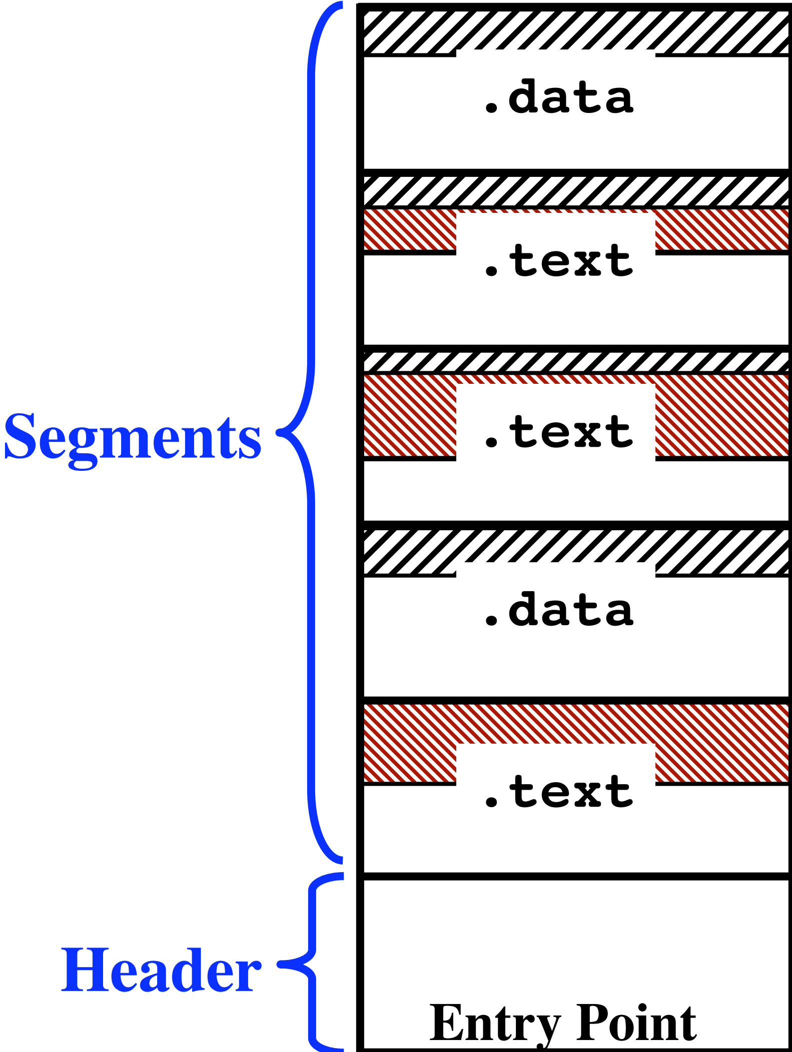
Cavity Viruses



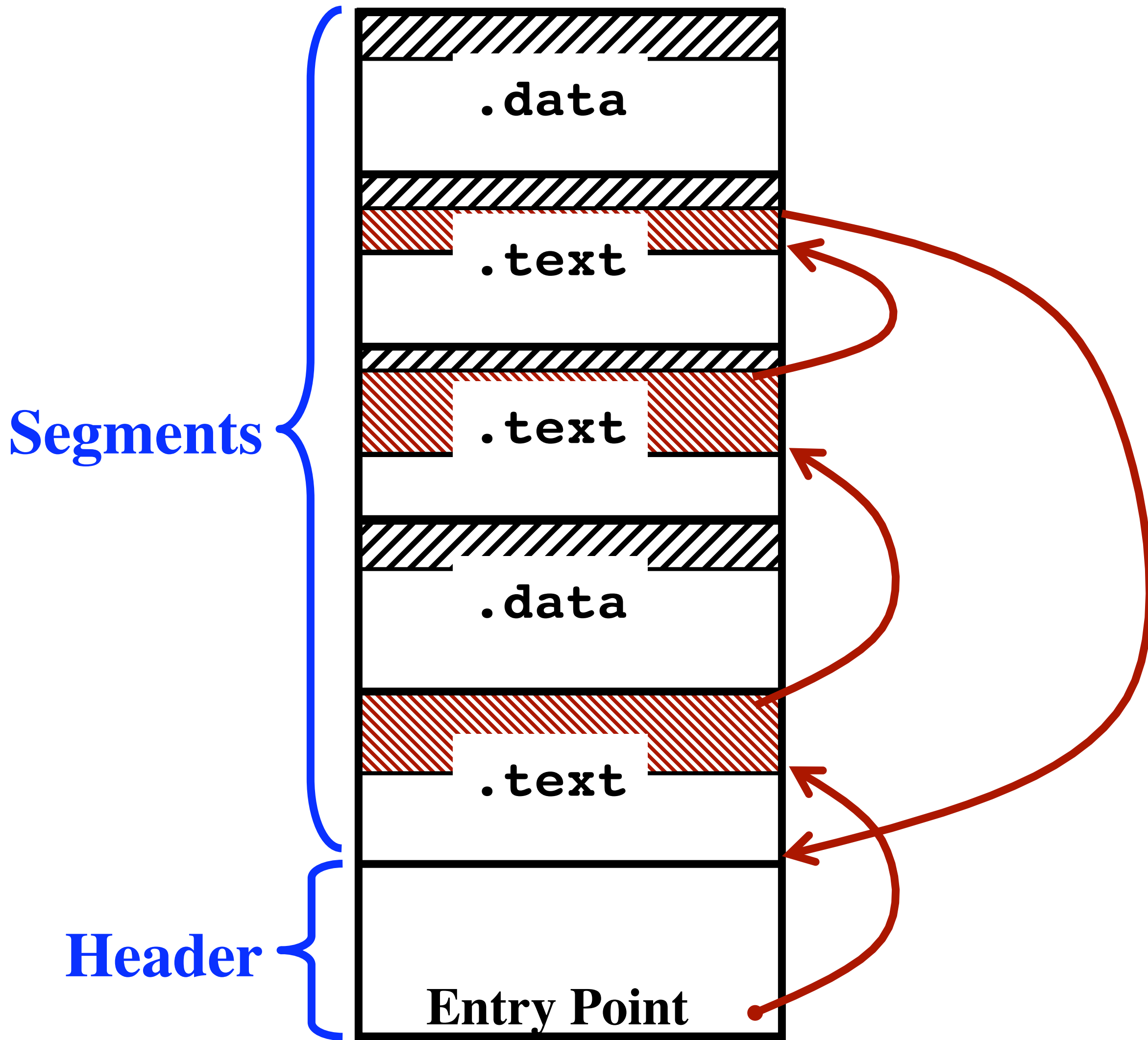
Cavity Viruses



Cavity Viruses



Cavity Viruses



Memory Resident Viruses

The virus stays in memory all the time.

  Does not exit when the infected program exits.

Virus hides in memory that is unused by the kernel

  Virus may modify kernel data structures

    E.g. Allocate a frame/buffer from the free pool

      to hold the virus code.

    Kernel is fooled & can't detect the missing memory.

Memory Resident Viruses

Virus must be executed periodically.
Idea: **"Capture an Interrupt Vector"**

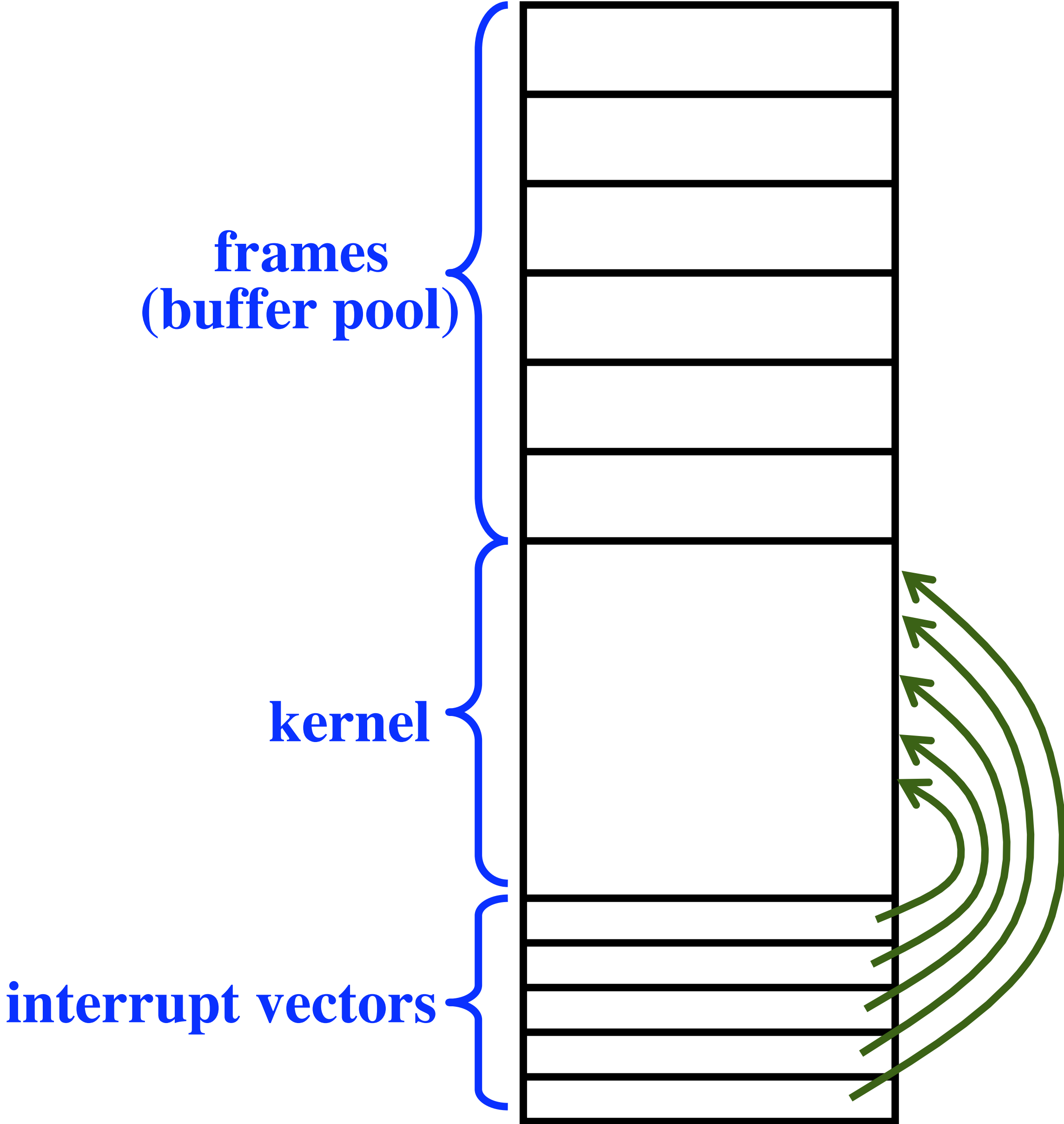
Recall:

- **An array of addresses in low memory.**
- **Each word is the address of one of the interrupt handlers.**
- **User process executes a SYSCALL.**
- **Hardware fetches an address from interrupt vector.**
- **Jumps to it.**

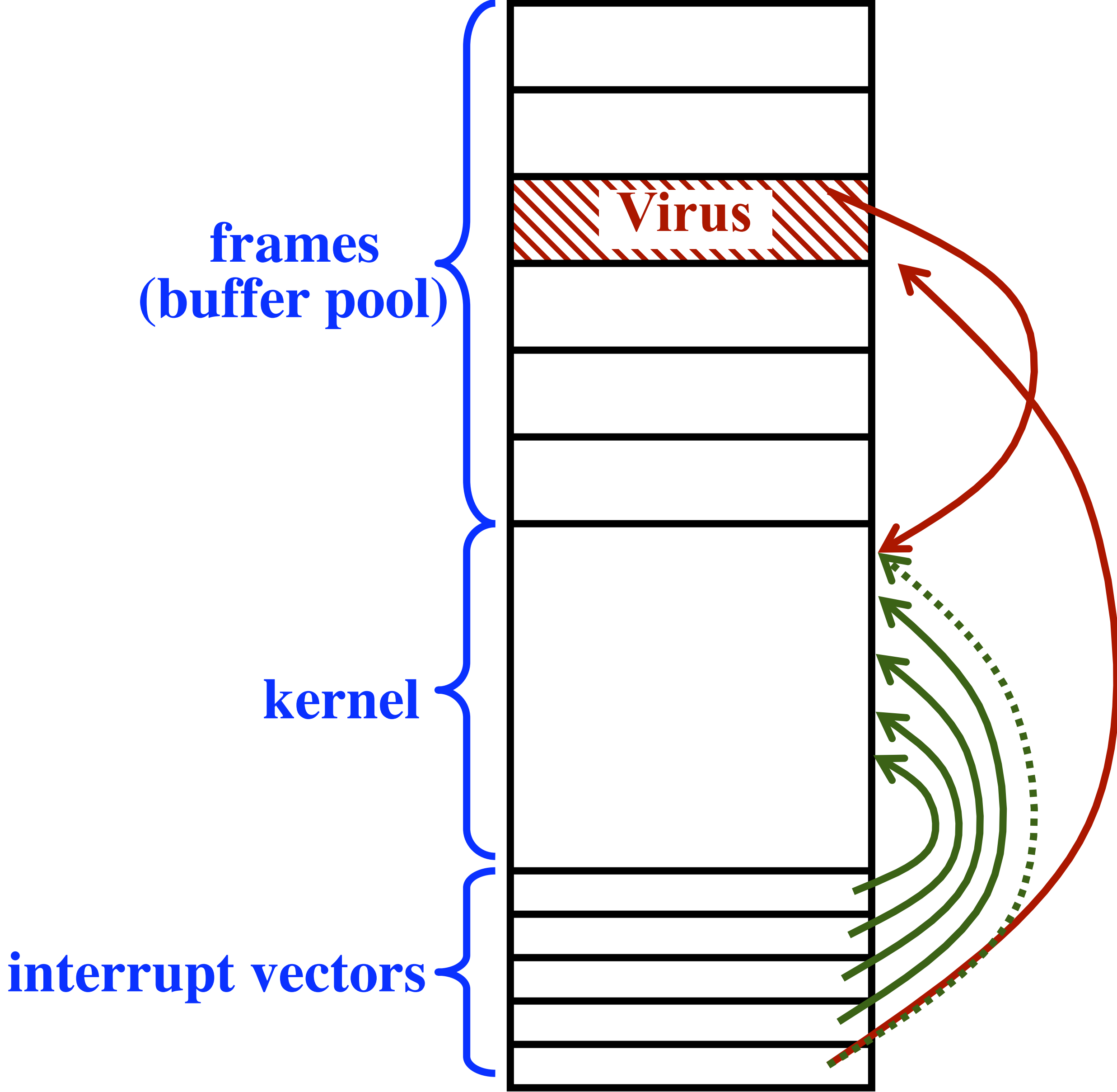
Virus:

- **Saves previous value of the interrupt vector.**
- **Overwrite, with address of virus.**
- **After virus executes, it jumps to normal handler.**
- **Virus runs everytime the interrupt occurs.**

Capturing an Interrupt Vector



Capturing an Interrupt Vector



Capturing an Interrupt Vector

Idea:

Virus runs on every SYSCALL.
Looks for "exec" call.
This is an excellent time to infect an executable!

Boot Sector Viruses

During booting...

BIOS reads Master Boot Record (MBR) from sector 0.

Loads it into RAM and jumps to it.

This program...

...determines which partition is the “boot partition”

...reads the boot sector from that partition.

...reads it into RAM and jumps to it.

This program loads a “kernel loader program.”

The loader reads in the entire kernel and jumps to it.

Boot Sector Viruses

MBR Viruses

The virus saves the original MBR (Master Boot Record) in some hidden sector (e.g., in a file or in a free sector)

The virus places its code in the MBR.

When the computer is booted the virus runs first.

- Virus copies itself to someplace else in memory (so the boot process will not overwrite the virus)
- CPU is in system mode & no anti-virus software active
- Virus can do its dirty-work!
- Virus reads in the true MBR and jumps to it.
  - Kernel starts up normally.

Boot Sector Viruses

Problem:

The virus will "lose control" of the CPU.

In Windows:

Does not load the entire interrupt vector at once.
Interrupts will occur while array is being loaded!

Idea:

Virus first captures all interrupt vectors.
Then virus jumps to the MBR and the kernel starts.
  On every interrupt...
    If the virus gets control,
      it re-captures all interrupt vectors!
After booting is complete,
  Virus is invoked on every interrupt / syscall!

Device Driver Viruses

The device driver code is loaded at boot-time.
The code can become infected.
(in the file containing the device driver executable code)
The virus is run whenever the driver is used / invoked.
Drivers usually run in “System Mode” as part of kernel.

Macro Viruses

Microsoft Word, Excel, Office

Users can write "macros"
Short programs that are executed upon a single keystroke.
They are interpreted (not compiled).
The macro is part of the document.
The macro can be attached to common commands
e.g., "Open File"

The Approach:

Bad guy emails an infected document.
The victim opens the MSWord attachment.
Opening the file causes the macro to be executed.
The macro can be an arbitrary Visual Basic program.

Source Code Viruses

The Infection Process:

Look for a “C” source code program file.

Insert virus code.

```
// comments
#include <stdio.h>

int i;
...
void MyFun (...) {
  ...
}
void main (...) {

  print (...);
  ...
}
```

Source Code Viruses

The Infection Process:

Look for a "C" source code program file.

Insert virus code.

```
// comments
#include <stdio.h>
#include <virus.h>
int i;
...
void MyFun (...) {
   ...
}
void main (...) {
  virus ();
  print (...);
   ...
}
```

Source Code Viruses

The Infection Process:

Look for a "C" source code program file.

Insert virus code.

```
// comments
#include <stdio.h>
#include <virus.h>
int i;
...
void MyFun (...) {
  ...
}
void main (...) {
  virus ();
  print (...);
  ...
}
```

Want to avoid discovery!

Source Code Viruses

The Infection Process:

Look for a "C" source code program file.

Insert virus code.

```
// comments
#include <stdio.h>
#include </usr/include/libxml/ctype.h>
int i;
...
void MyFun (...) {
  ...
}
void main (...) {
  init_type4spec ();
  print (...);
  ...
}
```

Want to avoid discovery!

Source Code Viruses

Must be able to parse and understand "C" code.
Will only work on machines where the included file exists.

Alternative:

- **Insert all the virus code into the program.**
- **How to avoid detection?**
  - **Compile the virus to machine code.**
    - **A sequence of hex bytes.**
  - **Insert the code as data.**

Browser Plug-in Viruses

User tries to view a webpage.
Browser says:

```
This page requires Plug-In "xyz".
 Do you want to install it?
```

The user...

- Installs a strange plug-in
- The browser opens the page
- The virus code executes

Email-Based Viruses

The email program automatically opens some attachments.
The attachment contains a virus.

Virus Mechanism:

The virus looks in the computer's address book.
Sends itself in email
to everybody in the address book
With an reasonable, interesting subject:
The "I love you" virus (2000) was very successful.

A New User...

Gets the infected email

- **Sender is someone they know**
- **The subject is interesting**
- **They naturally open the email**

Anti-Virus Techniques

Step 1: Get a copy of the virus.
Infect a "goat" file or computer

Step 2: Dis-assemble the virus code

Step 3: Understand how the code works

Step 4: Enter the code in a database of known viruses
If there are variations, look for a common pattern

The Anti-Virus Software...

- Download the list of virus patterns
- Look through all files
- If some file matches a pattern...
  Alert the user
  Delete the file
  Or modify the file to "break" the virus

Anti-Virus Techniques

Problem:

- **User has 100,000 files**
- **The virus database contains 100,000 patterns**
- **Scanning must be fast**
- **Viruses mutate themselves to increase the problem**
- **The match is “fuzzy” (not an exact match)**
- **False positives:**
  - **The scanner deletes a file unnecessarily**
  - **The scanner fails to detect a virus**

Anti-Virus Techniques

Idea:

Only check files that have changed.

Virus:

Virus avoids updating "time-last-modified".

Idea:

Look for files that have grown since the last scan.

Virus:

Contains code to compress and decompress data.
Virus compresses the host program.
Upon getting executed
- **The virus runs**
- **The virus decompress the host program**
- **The virus run the host code**

Encrypting Viruses

Anti-virus Software

Looks to see is data in a file matches a pattern.

Idea:

To avoid detection...
The virus encrypts itself.

Encryption / Decryption Algorithms

Usual goal: **Guarantee secrecy!**

Virus goal: **Make yourself look different each time!**
Makes life very hard on the anti-virus software.
No complex algorithm is needed.

Encrypting Viruses

How it works:

- **Virus contains:**
  - **Encryption function**
  - **Decryption function**
  - **Compression function**
  - **Decompression function**
  - **Payload code**
- **The virus compresses**
  - **Itself (except decryption, decompression functions)**
  - **The host program**
- **Virus encrypts itself (except the decryption function)**
- **When an infect file is executed:**
  - **Decryption stage runs first**
  - **Decompression stage runs second**

Encrypting Viruses

The entire virus is encrpyted

...except the decryption stage

Decryption stage can be very simple:

```
  mov   r1,address_of_virus
  mov   r2,number_of_bytes
loop:
  load  r3,[r1]
  xor   r3,0x357ad490,r3   ! Encryption key
  store [r1++],r3
  sub   r2,1,r2
  bne   loop
  jump  address_of_virus
```

Anti-virus scanners must look for the decryption stage.

Code Mutation Engines

Idea: Modify the code to disguise it.

- **Renumber the registers.**
- **Reorder instructions, where safe.**
- **Replace instructions with equivalent instructions**
- **Insert extra code that does nothing.**

"Polymorphic Viruses"

Mutate themselve each time they replicate

The anti-virus program...

Must look at 2 very different chunks of machine code.

Determine that they do the same thing.

(The "Halting Problem" - not decidable!)

Integrity Checking

Idea:

Anti-virus program runs before any files are infected.

It scans all files on the disk.

For each, it computes a checksum

Recall: SHA (Secure Hash Function)

MD5 (Message Digest)

It saves these in a file for later use.

Later, when it scans a file...

Checks to see if the file has changed.

Infected files will be caught.

Better protect the file with the checksum info from the virus!

Problem:

What if a file legitimately changes?

How to differentiate normal changes from infections?

Code Signing

Problem:

Want to download and run programs from the Internet

Need to know:

- The source: where did it come from?
- Who vouches for its security?
- Has it been modified?

Solution: Digital Signatures

The executable file contains:

(1) The code
(2) The source (e.g., URL of provider)
(3) The identity of the person vouching for it
(4) A digitial signature using public/private keys of (3).

Reliable, Secure, Trusted Operating Systems

Is it possible to build one?

Yes... (as long as there are no bugs in it!)

Why don't we have better security?

- Such an OS will not be compatible with existing systems
- Users require backward compatibility

Current challenges:

"Active content"

Files, webpages that contain executable code

Ease of downloading software

Increasing availability of software

Increasing complexity of software

The Future:

A really bad "viral incident"?