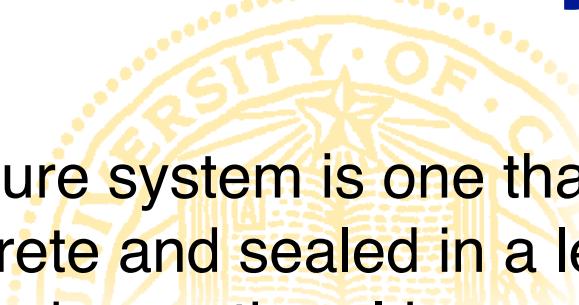


Security



The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards—and even then I have my doubts.

“Computer Recreations: Of Worms, Viruses and Core War”, A. K. Dewdney. In *Scientific American*, March 1989.

Security

- ❖ The security environment
 - ❖ Basics of cryptography
 - ❖ User authentication
 - ❖ Attacks from inside the system
 - ❖ Attacks from outside the system
 - ❖ Protection mechanisms
 - ❖ Trusted systems
-
- ❖ If you're interested in this stuff, consider taking CMPS 122 in Winter 2013

Security environment: threats

Goal	Threat
Data confidentiality	Exposure of data
Data integrity	Tampering with data
System availability	Denial of service

- ❖ Operating systems have goals
 - Confidentiality
 - Integrity
 - Availability
- ❖ Someone attempts to subvert the goals
 - Fun
 - Commercial gain

What kinds of intruders are there?

- ❖ Casual prying by nontechnical users
 - Curiosity
- ❖ Snooping by insiders
 - Often motivated by curiosity or money
- ❖ Determined attempt to make money
 - May not even be an insider
- ❖ Determined attempt to make mischief
 - Money typically not a goal
 - Inconvenience others or prove a point
- ❖ Commercial or military espionage
 - This is very big business!

Accidents cause problems, too...

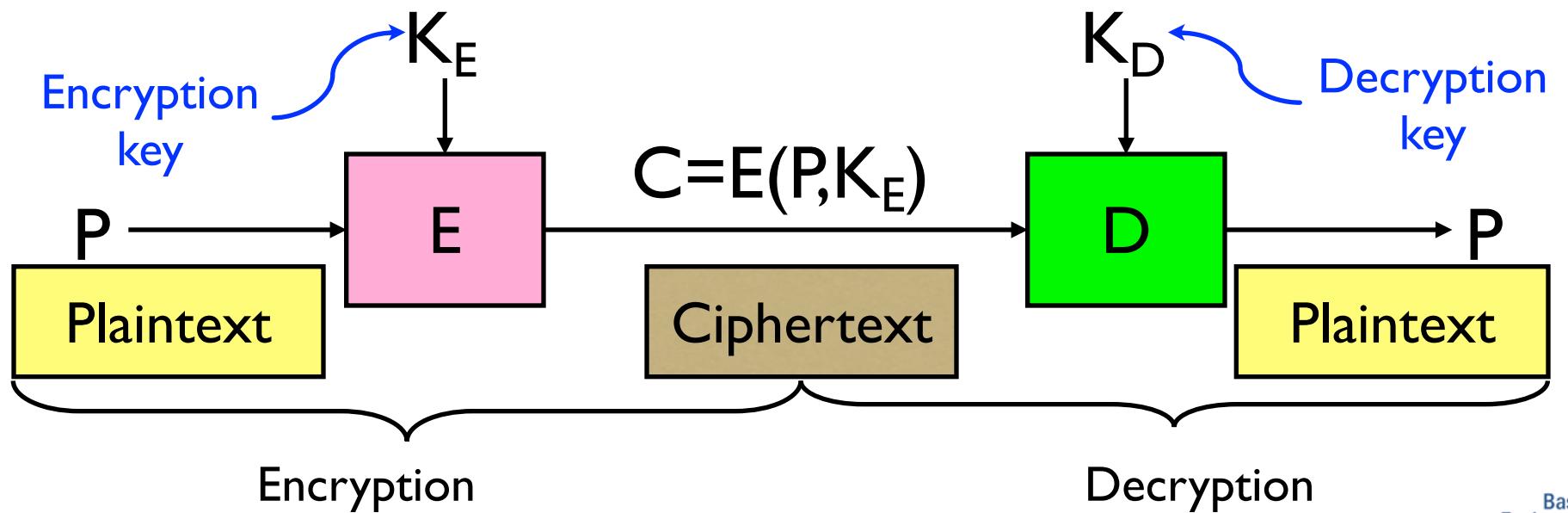
- ♦ Acts of God
 - Fires
 - Earthquakes
 - Wars (is this really an “act of God”?)
- ♦ Hardware or software error
 - CPU malfunction
 - Disk crash
 - Program bugs (hundreds of bugs found in the most recent Linux kernel)
- ♦ Human errors
 - Data entry
 - Wrong tape mounted
 - `rm * .o`

Cryptography

- ✿ Goal: keep information from those who aren't supposed to see it
 - Do this by “scrambling” the data
- ✿ Use a well-known algorithm to scramble data
 - Algorithm has two inputs: data & key
 - Key is known only to “authorized” users
 - Relying upon the secrecy of the algorithm is a very bad idea (see WW2 Enigma for an example...)
- ✿ Cracking codes is very difficult, *Sneakers* and *Swordfish* (and other movies) notwithstanding

Cryptography basics

- Algorithms (E, D) are widely known
- Keys (K_E, K_D) should be less widely distributed
- For this to be effective, the **ciphertext** should be the only information that's available to the world
- **Plaintext** is known only to the people with the keys (in an ideal world...)



Secret-key encryption

- ♦ Also called symmetric-key encryption
- ♦ Monoalphabetic substitution
 - Each letter replaced by different letter

- ♦ Vignere cipher
 - Use a multi-character key

THEMESSAGE

ELMELMELME

XSQQPEWLSI

- ♦ Both are easy to break!
- ♦ Given the encryption key, easy to generate the decryption key
- ♦ Alternatively, use different (but similar) algorithms for encryption and decryption

Modern encryption algorithms

- ✿ Data Encryption Standard (DES)
 - Uses 56-bit keys
 - Same key is used to encrypt & decrypt
 - Keys used to be difficult to guess
 - Needed to try 2^{55} different keys, on average
 - Modern computers can try millions of keys per second with special hardware
 - For \$250K, EFF built a machine that broke DES quickly
- ✿ Current algorithms (AES, Blowfish) use at least 128 bit keys
 - Adding one bit to the key makes it twice as hard to guess
 - Must try 2^{127} keys, on average, to find the right one
 - At 10^{15} keys per second, this would require over 10^{21} seconds, or 1000 billion years!
 - Modern encryption isn't usually broken by brute force...

Unbreakable codes

- ✿ There is such a thing as an unbreakable code: one-time pad
 - Use a **truly random** key as long as the message to be encoded
 - XOR the message with the key a bit at a time
- ✿ Code is unbreakable because
 - Key could be anything
 - Without knowing key, message could be anything with the correct number of bits in it
- ✿ Difficulty: distributing key is as hard as distributing message
 - May be easier because of timing
- ✿ Difficulty: generating truly random bits
 - Can't use computer random number generator!
 - May use physical processes
 - Radioactive decay
 - Leaky diode
 - Lava lamps (!): <http://www.sciencenews.org/20010505/mathtrek.asp>
 - Webcams (with lens cap on): <http://www.lavarnd.org/>

Public-key cryptography

- ✿ Instead of using a single shared secret, keys come in pairs
 - One key of each pair distributed widely (*p**U**blic key*), K_U
 - One key of each pair kept secret (*p**R**ivate or secret key*), K_R
 - Keys are inverses of one another, but not identical
 - Encryption & decryption are the same algorithm, so
$$E(K_U, E(K_R, M)) = E(K_R, E(K_U, M)) = M$$
- ✿ Currently, the most popular method involves primes and exponentiation
 - Difficult to crack unless large numbers can be factored
 - Very slow for large messages

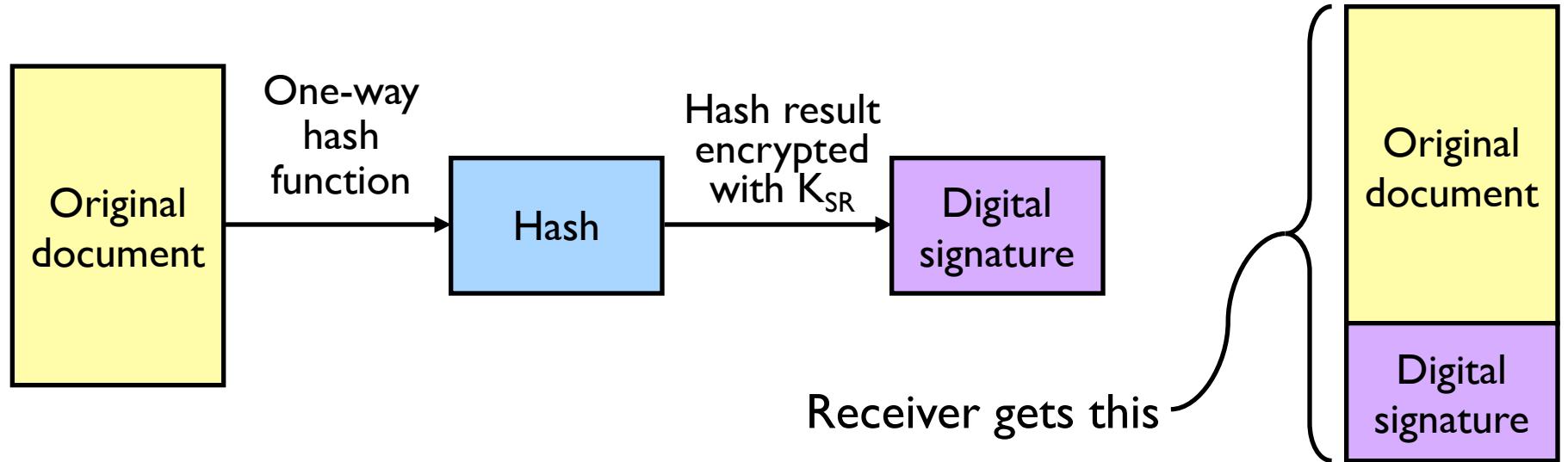
RSA algorithm for public key encryption

- ❖ Private, public key pair consists of $K_R = (d, n)$ $K_U = (e, n)$
 - $n = p \times q$ (p and q are large prime numbers)
 - e is a randomly chosen integer with $\text{GCD}(e, (p-1) \times (q-1)) = 1$
 - d is an integer such that $(e \times d) \text{ MOD } ((p-1) \times (q-1)) = 1$
- ❖ p & q aren't published, and it's hard to compute them: factoring large numbers is “difficult”
- ❖ Public key is published, and can be used by anyone to send a message to the private key's owner
- ❖ Encryption & decryption are the same algorithm:
 $E(K_U, M) = M^e \text{ MOD } n$ (similar for K_R)
 - Methods exist for doing the above calculation quickly, but...
 - Exponentiation is still very slow

One-way functions

- ♦ Function such that
 - Given formula for $f(x)$, easy to evaluate $y = f(x)$
 - Given y , computationally infeasible to find any x such that $y = f(x)$
- ♦ Often, operate similar to encryption algorithms
 - Produce fixed-length output rather than variable length output
 - Similar to XOR-ing blocks of ciphertext together
- ♦ Common algorithms include
 - MD5: 128-bit result
 - SHA-1: 160-bit result

Digital signatures



- ❖ Digital signature computed by
 - Applying one-way hash function to original document
 - Encrypting result with **sender's private key**
- ❖ Receiver can verify by
 - Applying one-way hash function to received document
 - Decrypting signature using sender's public key
 - Comparing the two results: equality means document unaltered

Pretty Good Privacy (PGP)

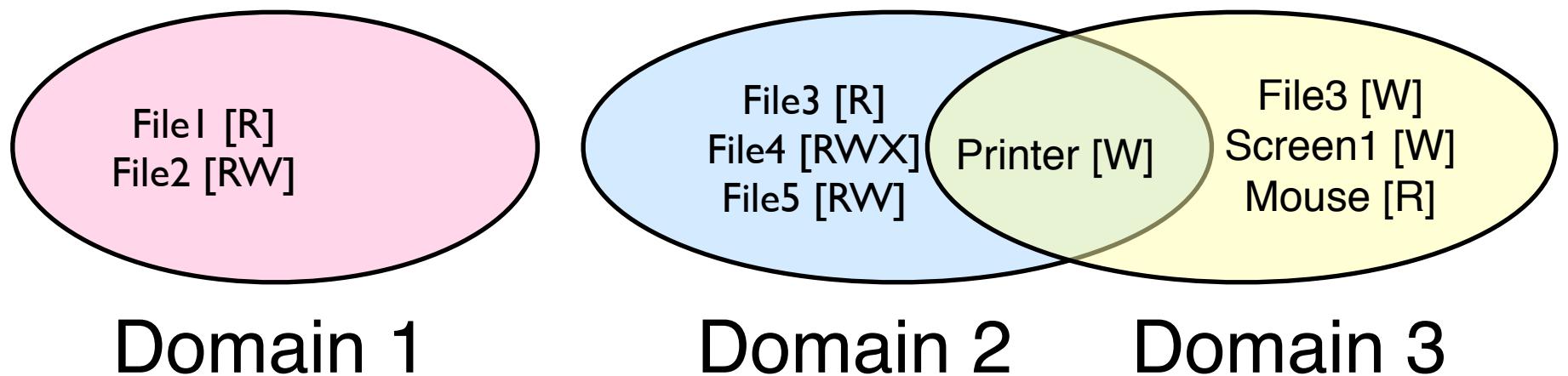
- ✿ Uses public key encryption
 - Facilitates key distribution
 - Allows messages to be sent encrypted to a person (encrypt with person's public key)
 - Allows person to send message that must have come from her (encrypt with person's private key)
- ✿ Problem: public key encryption is very slow
- ✿ Solution: use public key encryption to exchange a shared key
 - Shared key is relatively short (~128 bits)
 - Message encrypted using symmetric key encryption
- ✿ PGP can also be used to authenticate sender
 - Use digital signature and send message as plaintext

Protection

- ❖ Security is mostly about **mechanism**
 - How to enforce policies
 - Policies largely independent of mechanism
- ❖ Protection is about specifying **policies**
 - How to decide who can access what?
- ❖ Specifications must be
 - Correct
 - Efficient
 - Easy to use (or nobody will use them!)

Protection domains

- ♦ Three protection domains
 - Each lists objects with permitted operations
- ♦ Domains can share objects & permissions
 - Objects can have different permissions in different domains
 - There need be no overlap between object permissions in different domains
- ♦ How can this arrangement be specified more formally?



Protection matrix

Domain	File1	File2	File3	File4	File5	Printer1	Mouse
1	Read	Read Write					
2			Read	Read Write Execute	Read Write	Write	
3			Write		Write	Read	

- Each domain has a row in the matrix
- Each object has a column in the matrix
- Entry for <object,column> has the permissions
- Who's allowed to modify the protection matrix?
 - What changes can they make?

Domains: objects in the protection matrix

Domain	File1	File2	File3	File4	Printer1	Mouse	Dom1	Dom2	Dom3
1	Read	Read Write					Modify		
2		Read	Read Write Execute		Write		Modify		
3		Write			Write	Read		Enter	

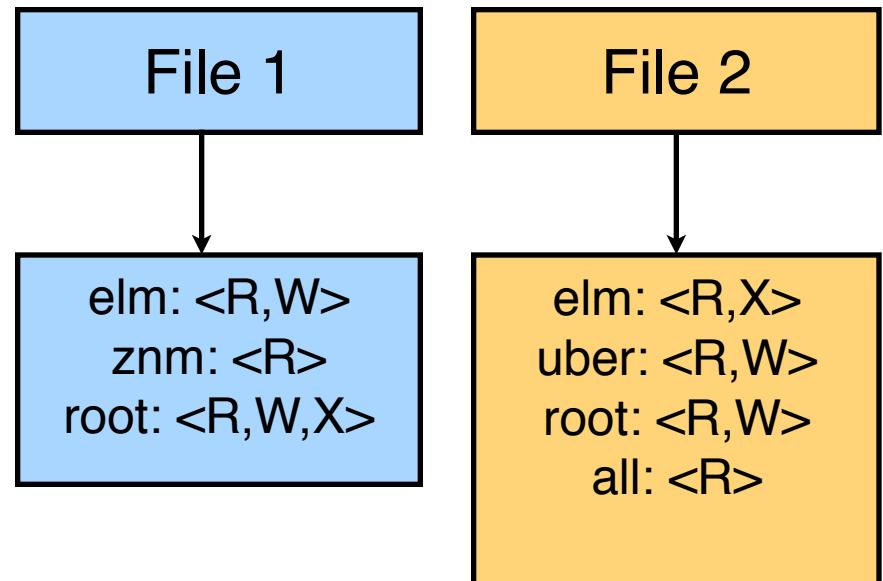
- Specify permitted operations on domains in the matrix
 - Domains may (or may not) be able to modify themselves
 - Domains can modify other domains
 - Some domain transfers permitted, others not
- Doing this allows flexibility in specifying domain permissions
 - Retains ability to restrict modification of domain policies

Representing the protection matrix

- ♦ Need to find an efficient representation of the protection matrix (also called the access matrix)
- ♦ Most entries in the matrix are empty!
- ♦ Compress the matrix by:
 - Associating permissions with each object: **access control list**
 - Associating permissions with each domain: **capabilities**
- ♦ How is this done, and what are the tradeoffs?

Access control lists

- ♦ Each object has a list attached to it
- ♦ List has
 - Protection domain
 - User name
 - Group of users
 - Other
 - Access rights
 - Read
 - Write
 - Execute (?)
 - Others?
- ♦ No entry for domain ⇒ no rights for that domain
- ♦ Operating system checks permissions when access is needed

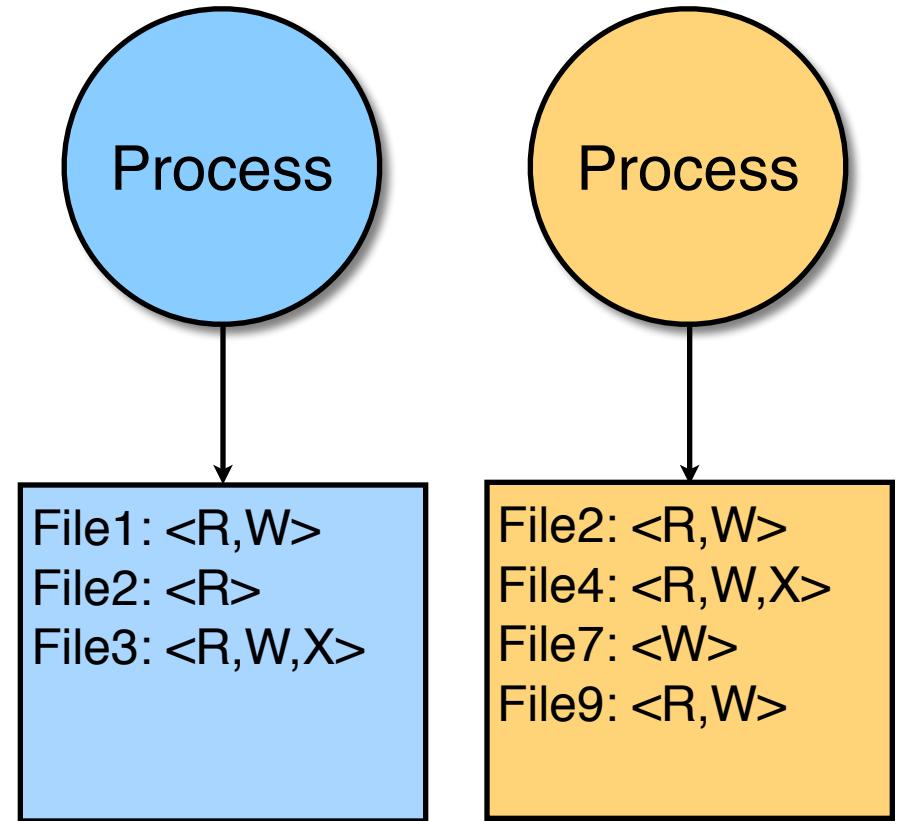


Access control lists in the real world

- ♦ Unix file system
 - Access list for each file has exactly three domains on it
 - User (owner)
 - Group: set of users
 - Others
 - Rights include read, write, execute: interpreted differently for directories and files
 - Users may be in more than one group
- ♦ AFS (unix.ic)
 - Access lists only apply to directories: files inherit rights from the directory they're in
 - Access list may have many entries on it with possible rights:
 - read, write, lock (for files in the directory)
 - lookup, insert, delete (for the directories themselves),
 - administer (ability to add or remove rights from the ACL)

Capabilities

- ♦ Each process has a capability list
- ♦ List has one entry per object the process can access
 - Object name
 - Object permissions
- ♦ Objects not listed are not accessible
- ♦ How are these secured?
 - Kept in kernel
 - Cryptographically secured



Cryptographically protected capability

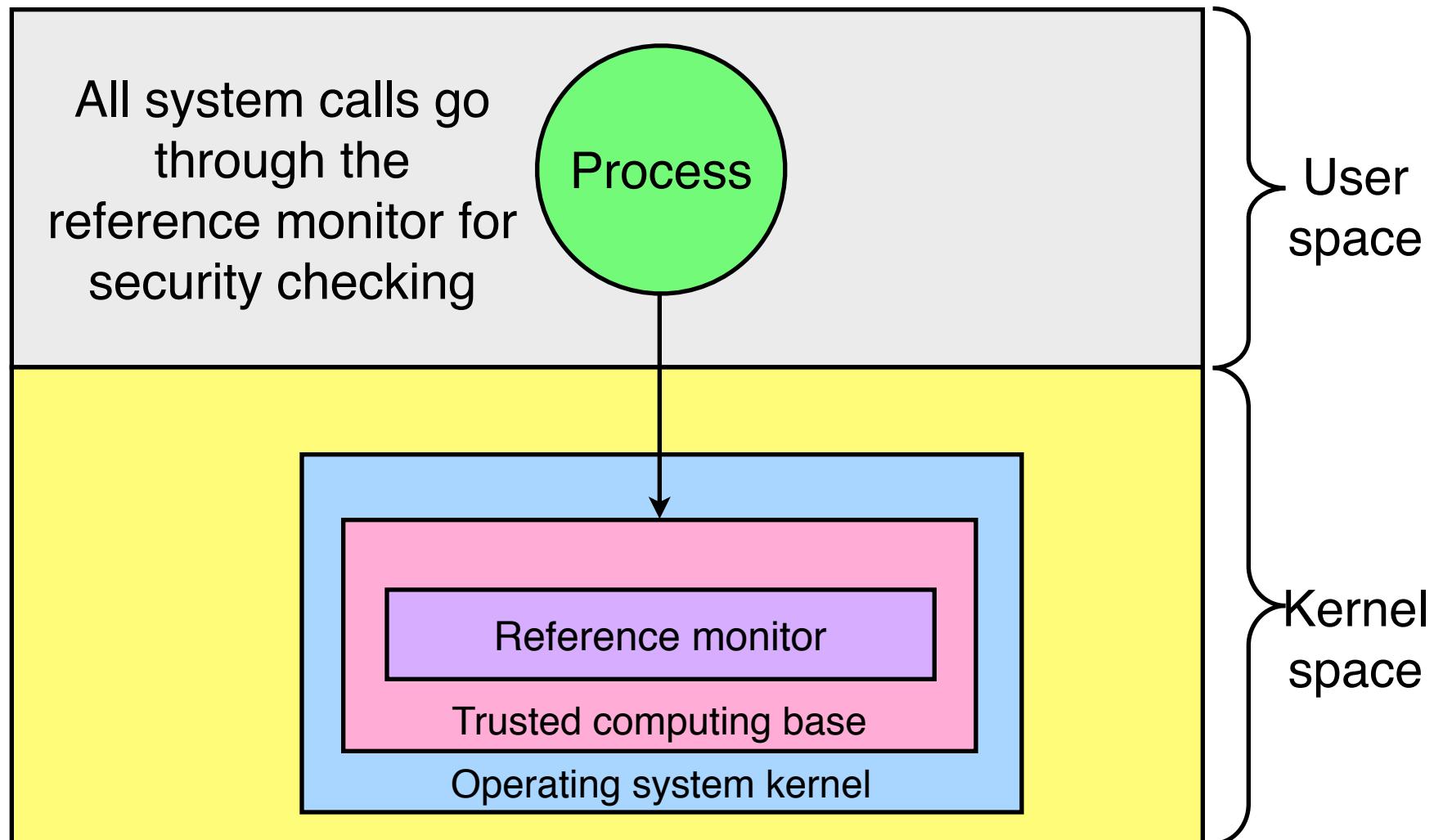


- ✿ Rights include generic rights (read, write, execute) and
 - Copy capability
 - Copy object
 - Remove capability
 - Destroy object
- ✿ Server has a secret (Check) and uses it to verify capabilities presented to it
 - Alternatively, use public-key signature techniques

Protecting the access matrix: summary

- ♦ OS must ensure that the access matrix isn't modified (or even accessed) in an unauthorized way
- ♦ Access control lists
 - Reading or modifying the ACL is a system call
 - OS makes sure the desired operation is allowed
- ♦ Capability lists
 - Can be handled the same way as ACLs: reading and modification done by OS
 - Can be handed to processes and verified cryptographically later on
 - May be better for widely distributed systems where capabilities can't be centrally checked

Reference monitor

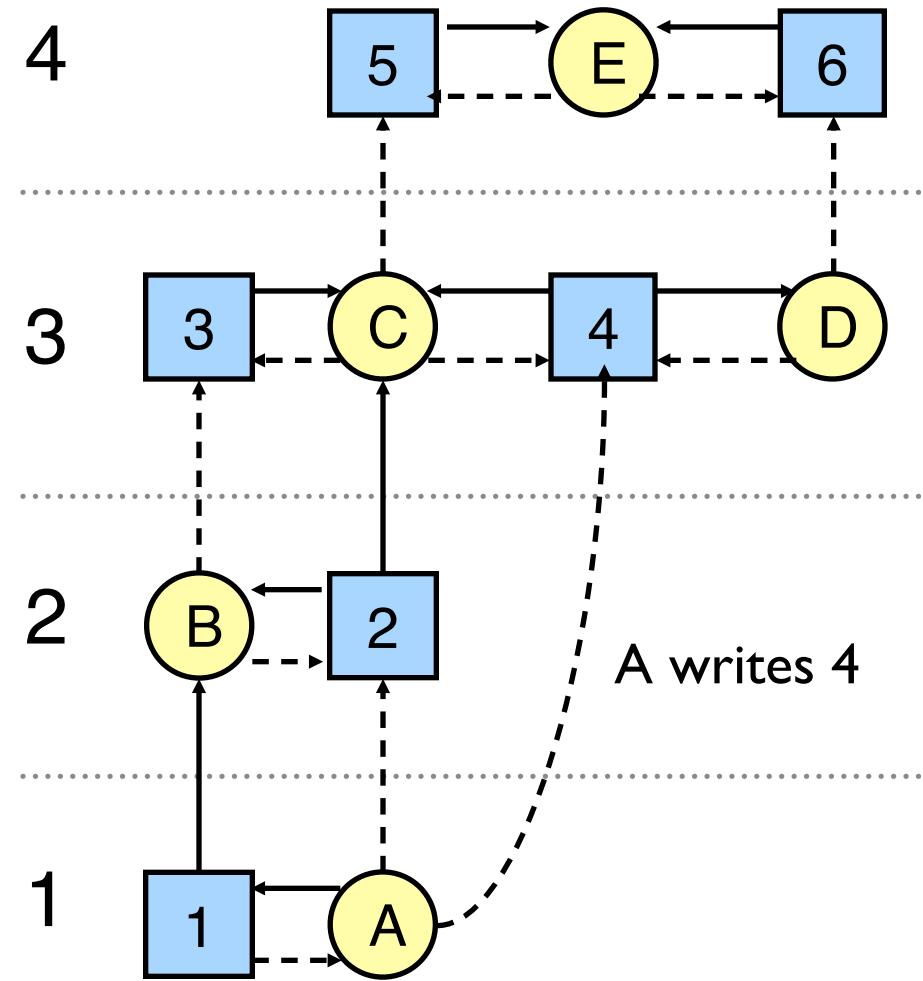


Formal models of secure systems

- ❖ Limited set of primitive operations on access matrix
 - Create/delete object
 - Create/delete domain
 - Insert/remove right
- ❖ Primitives can be combined into **protection commands**
 - May not be combined arbitrarily!
- ❖ OS can enforce policies, but can't decide what policies are appropriate
- ❖ Question: is it possible to go from an “authorized” matrix to an “unauthorized” one?
 - In general, undecidable
 - May be provable for limited cases

Bell-La Padula multilevel security model

- Processes, objects have security level
- Simple security property
 - Process at level k can only read objects at levels k or lower
- * property
 - Process at level k can only write objects at levels k or higher
- These prevent information from leaking from higher levels to lower levels



Biba multilevel integrity model

- ❖ Principles to guarantee integrity of data
- ❖ Simple integrity principle
 - A process can write only objects at its security level or lower
 - No way to plant fake information at a higher level
- ❖ The **integrity *** property
 - A process can read only objects at its security level or higher
 - Prevent someone from getting information from above and planting it at their level
- ❖ Biba is in direct conflict with Bell-La Padula
 - Difficult to implement both at the same time!

Covert channels

- ♦ Circumvent security model by using more subtle ways of passing information
- ♦ Can't directly send data against system's wishes
- ♦ Send data using “side effects”
 - Allocating resources
 - Using the CPU
 - Locking a file
 - Making small changes in legal data exchange
- ♦ Very difficult to plug leaks in covert channels!

Covert channel using file locking

- ❖ Exchange information using file locking
- ❖ Assume $n+1$ files accessible to both A and B
- ❖ A sends information by
 - Locking files $0..n-1$ according to an n -bit quantity to be conveyed to B
 - Locking file n to indicate that information is available
- ❖ B gets information by
 - Reading the lock state of files $0..n+1$
 - Unlocking file n to show that the information was received
- ❖ May not even need access to the files (on some systems) to detect lock status!

Steganography

- ❖ Hide information in other data
- ❖ Picture on right has text of 5 Shakespeare plays
 - Encrypted, inserted into low order bits of color values



Zebras



Hamlet, Macbeth, Julius Caesar
Merchant of Venice, King Lear

User authentication

- ❖ Problem: how does the computer know who you are?
- ❖ Solution: use **authentication** to identify
 - Something the user knows
 - Something the user has
 - Something the user is
- ❖ This must be done before user can use the system
- ❖ Important: from the computer's point of view...
 - Anyone who can duplicate your ID **is** you
 - Fooling a computer isn't all that hard...

Authentication using passwords

Login: **elm**

Password: **foobar**

Welcome to Linux!

Login: **mstorer**

User not found!

Login:

Login: **elm**

Password: **barfle**

Invalid password!

Login:

- ♦ Successful login lets the user in
- ♦ If things don't go so well...
 - Login rejected after name entered
 - Login rejected after name and incorrect password entered
- ♦ Don't notify the user of incorrect user name until after the password is entered!
 - Early notification can make it easier to guess valid user names

Dealing with passwords

- ✿ Passwords should be memorable
 - Users shouldn't need to write them down!
 - Users should be able to recall them easily
- ✿ Passwords shouldn't be stored “in the clear”
 - Password file is often readable by all system users!
 - Password must be checked against entry in this file
- ✿ Solution: use hashing to hide “real” password
 - One-way function converting password to meaningless string of digits (Unix password hash, MD5, SHA-1)
 - Difficult to find another password that hashes to the same random-looking string
 - Knowing the hashed value and hash function gives no clue to the original password

Salting the passwords

- ✿ Passwords can be guessed
 - Before starting, build a table of all dictionary words, names, etc.
 - Table has each potential password in both plain and hashed form
 - Hackers can get a copy of the password file
 - For each entry in the password file, see if the password is in the above table
 - If it is, you have a password: works on more passwords than you'd think!
- ✿ Solution: use “salt”
 - Random characters added to the password before hashing
 - Salt characters stored “in the clear”
 - Increase the number of possible hash values for a given password
 - Actual password is “pass”
 - Salt = “aa” → hash (“passaa”)
 - Salt = “bb” → hash (“passbb”)
 - Result: cracker has to try many more combinations
- ✿ Mmmm, salted passwords!

One-time passwords

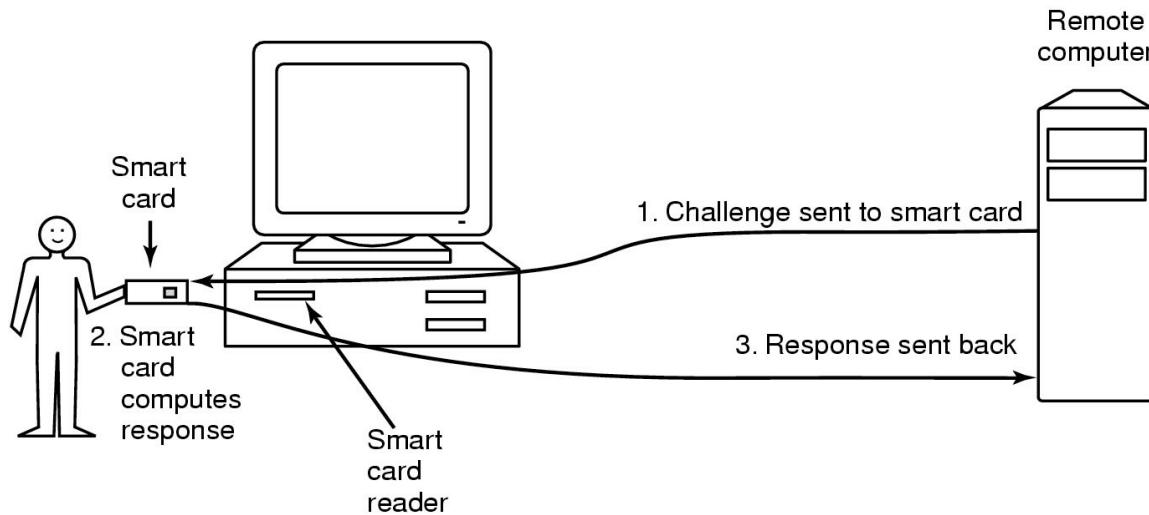
- ♦ Often useful to have passwords that can't be reused
 - User has a list of passwords to use
 - Each password is invalid after being used once
- ♦ Typically implemented using one-way hash chain
 - Password n is calculated by doing $f^n(s)$, where s is a secret and f is a one-way function
 - Hash s a total of n times for password n
 - Easy to calculate previously-used passwords
 - Difficult to calculate the next password in the list
- ♦ Server is initialized with the list of passwords
- ♦ On each login, server passes n to the client, which hashes the password n times and returns the answer
 - Since n is decremented each time, attacker can't compute the next password
 - Need to reinitialize after n password uses....

Sample breakin (from LBL)

```
LBL> telnet elksi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

Moral: change all the default system passwords!

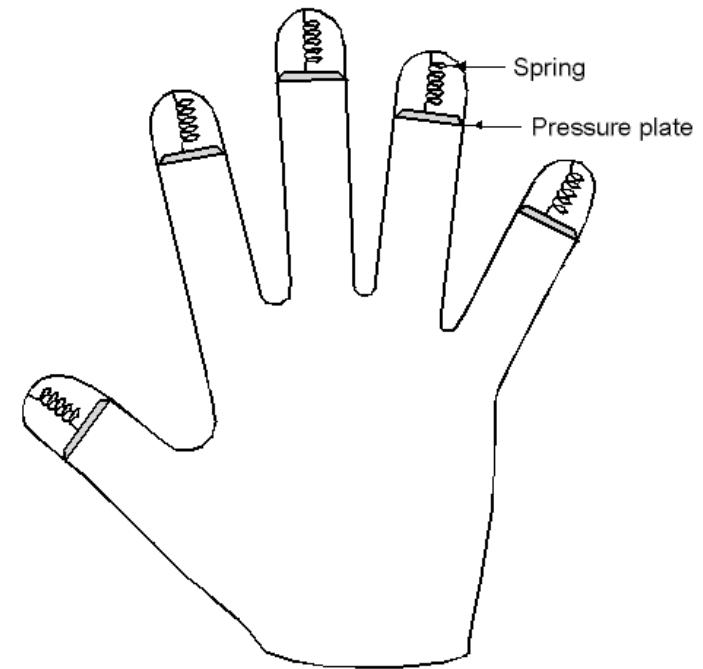
Authentication using a physical object



- ❖ Magnetic card
 - Stores a password encoded in the magnetic strip
 - Allows for longer, harder to memorize passwords
- ❖ Smart card
 - Card has secret encoded on it, but not externally readable
 - Remote computer issues challenge to the smart card
 - Smart card computes the response and proves it knows the secret

Authentication using biometrics

- ♦ Use basic body properties to prove identity
- ♦ Examples include
 - Fingerprints
 - Voice
 - Hand size
 - Retina patterns
 - Iris patterns
 - Facial features
- ♦ Potential problems
 - Duplicating the measurement
 - Stealing it from its original owner?



Countermeasures

- ❖ Limiting times when someone can log in
- ❖ Automatic callback at number prespecified
 - Can be hard to use unless there's a modem involved
- ❖ Limited number of login tries
 - Prevents attackers from trying lots of combinations quickly
- ❖ A database of all logins
- ❖ Simple login name/password as a trap
 - Security personnel notified when attacker bites
 - Variation: allow anyone to "log in," but don't let intruders do anything useful

Attacks on computer systems

- ❖ Insider attacks

- Logic bombs
 - Trap doors
 - Login spoofing

- ❖ Exploiting bugs in code

- ❖ Malware

- Trojan horses
 - Viruses

Logic bombs

- ♦ Programmer writes (complex) program
 - Wants to ensure that he's treated well
 - Embeds logic "flaws" that are triggered if certain things aren't done
 - Enters a password daily (weekly, or whatever)
 - Adds a bit of code to fix things up
 - Provides a certain set of inputs
 - Programmer's name appears on payroll (really!)
- ♦ If conditions aren't met
 - Program simply stops working
 - Program may even do damage
 - Overwriting data
 - Failing to process new data (and not notifying anyone)
- ♦ Programmer can blackmail employer
- ♦ Needless to say, this is highly unethical!

Trap doors

```
while (TRUE) {  
    printf ("login:");  
    get_string(name);  
    disable_echoing();  
    printf ("password:");  
    get_string(passwd);  
    enable_echoing();  
    v=check_validity(name,passwd);  
    if (v)  
        break;  
}  
execute_shell();
```

Normal code

```
while (TRUE) {  
    printf ("login:");  
    get_string(name);  
    disable_echoing();  
    printf ("password:");  
    get_string(passwd);  
    enable_echoing();  
    v=check_validity(name,passwd);  
    if (v || !strcmp(name, "elm"))  
        break;  
}  
execute_shell();
```

Code with trapdoor

Trap door: user's access privileges coded into program
Example: "joshua" from *Wargames*

Login spoofing



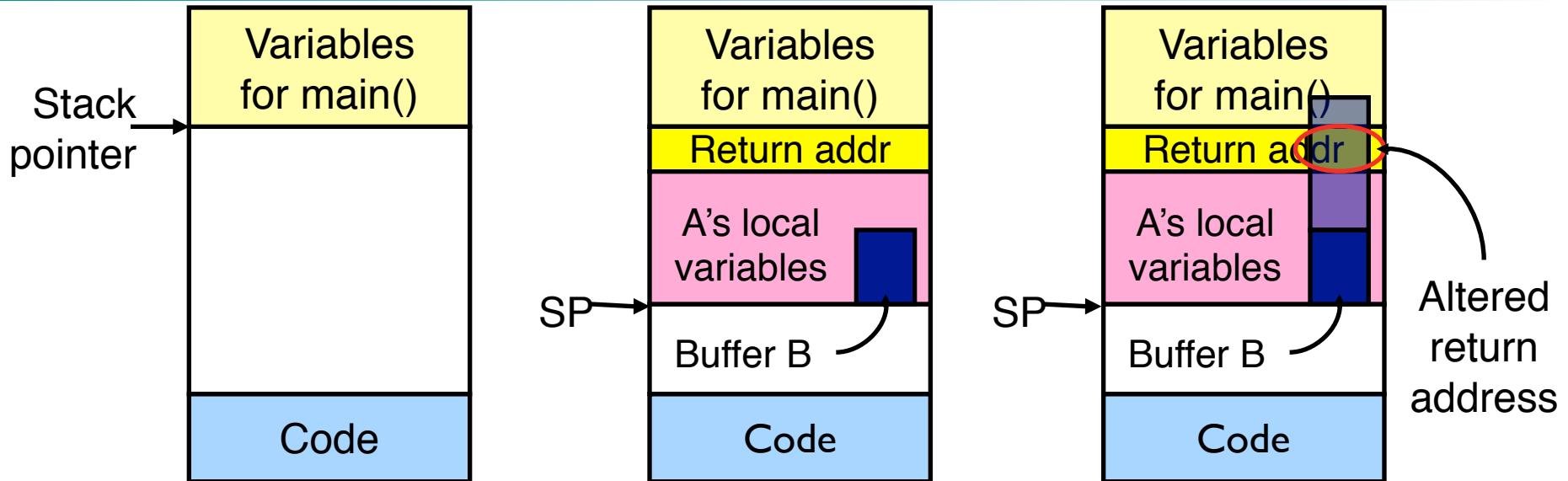
Real login screen



Phony login screen

- ❖ No difference between real & phony login screens
- ❖ Intruder sets up phony login, walks away
- ❖ User logs into phony screen
 - Phony screen records user name, password
 - Phony screen prints “login incorrect” and starts real screen
 - User retypes password, thinking there was an error
- ❖ Solution: don’t allow certain characters to be “caught”

Buffer overflow



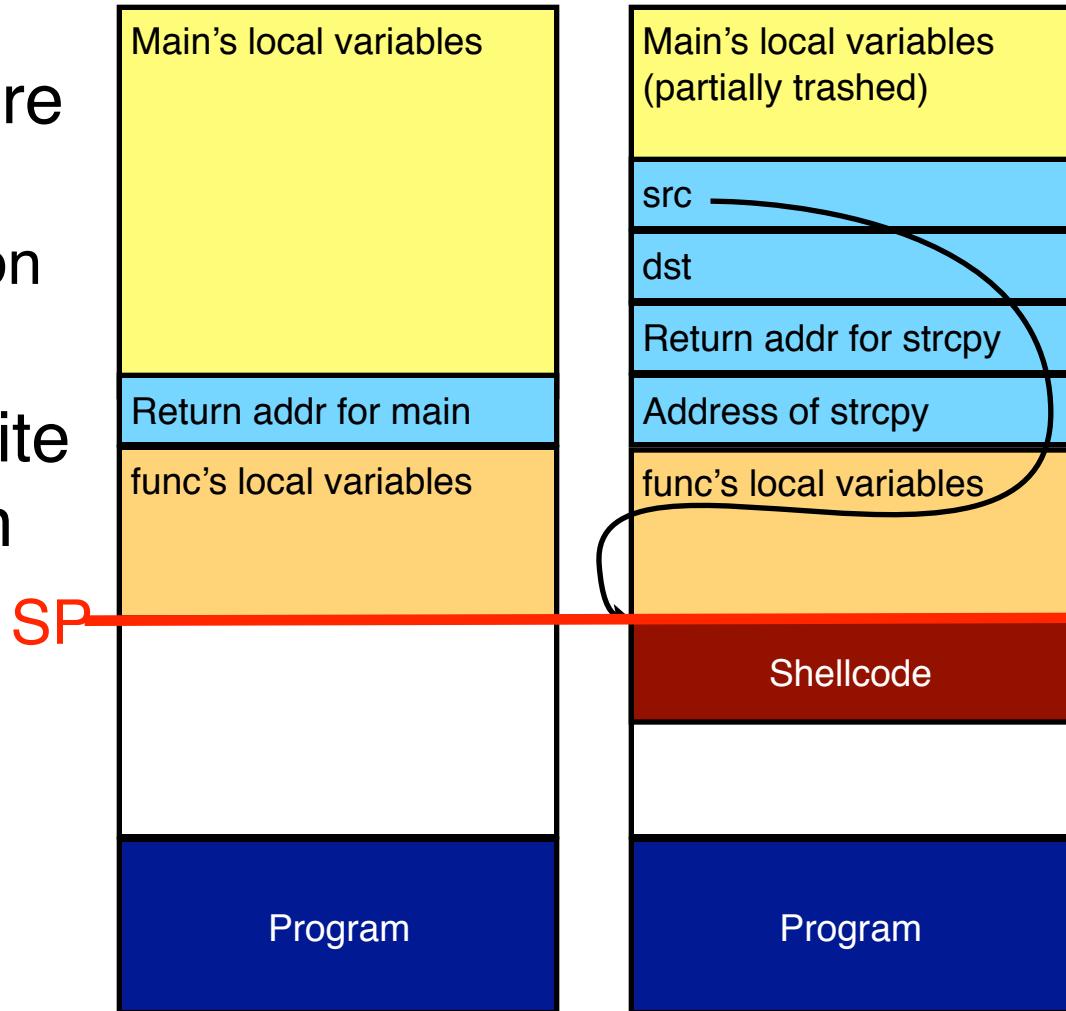
- ✿ Buffer overflow is a big source of bugs in operating systems
 - Most common in user-level programs that help the OS do something
 - May appear in “trusted” daemons
- ✿ Exploited by modifying the stack to
 - Return to a different address than that intended
 - Include code that does something malicious
- ✿ Accomplished by writing past the end of a buffer on the stack

Format string attacks

- `printf()` can take a variable for the format string
 - `format_str = "%s %d\n"`
`printf(format_str, arg1, arg2)`
- Overwrite `format_str` and use the `%n` operator in the string: writes string length so far to next arg
- Can be very effective if arguments can be controlled by the attacker
 - Inputs on the command line
 - Inputs on web forms

Return to libc attack

- ♦ Format string and buffer overflow attacks may require executable code on stack
 - Simple fix: don't allow data on stack to be executed!
- ♦ Variation on attack: overwrite stack to return to a location in libc
 - Doesn't require executable stack!
- ♦ Set up stack so that `strcpy()` is called to copy bytes to the data segment and run there



Code injection attacks

- ♦ Many programs call the OS to do something useful
 - `system()`
- ♦ Others use strings directly in database commands
- ♦ Attack: specify arguments that tack on additional functionality when copied into the command string

```
char src[100], dst[100], cmd[300];
getarg (src);
getarg (dst);
sprintf (cmd, "cp %s %s");
system (cmd);
```

What happens if:

`src = "abc"`

`dst = "xyz"`

`dst = "xyz ; rm -rf /home/elm"`

Generic security attacks

- ❖ Request memory, disk space, tapes and just read
- ❖ Try illegal system calls
- ❖ Start a login and hit DEL, RUBOUT, or BREAK
- ❖ Try modifying complex OS structures
- ❖ Try to do specified DO NOTs
- ❖ Social engineering
 - Convince a system programmer to add a trap door
 - Beg admin's secretary (or other people) to help a poor user who forgot password
 - Pretend you're tech support and ask random users for their help in debugging a problem

Security in a networked world

- ❖ External threat
 - Code transmitted to target machine
 - Code executed there, doing damage
- ❖ Goals of virus writer
 - Quickly spreading virus
 - Difficult to detect
 - Hard to get rid of
 - Optional: does something malicious
- ❖ Virus: embeds itself into other (legitimate) code to reproduce and do its job
 - Attach its code to another program
 - Additionally, may do harm

Malware

- ❖ Programs that attempt to cause damage
 - Goal: “infect” the target machine
- ❖ Types of damage
 - Denial of service
 - Permanently damage data (or even hardware!)
 - Espionage
 - Practical joke (not much damage)
 - Blackmail
- ❖ Terminology
 - *backdoor*: allows malware creator to control target machine
 - *zombie*: machine that’s been taken over
 - *botnet*: collection of zombies

Trojan horses

- ♦ Free program made available to unsuspecting user
 - Actually contains code to do harm
 - May do something useful as well...
- ♦ Altered version of utility program on victim's computer
 - Trick user into running that program
- ♦ Example (getting superuser access on unix.ic?)
 - Place a file called `ls` in your home directory
 - File creates a shell in `/tmp` with privileges of whoever ran it
 - File then actually runs the real `ls`
 - Complain to your sysadmin that you can't see any files in your directory
 - Sysadmin runs `ls` in your directory
 - Hopefully, he runs `your` `ls` rather than the real one (depends on his search path)

How viruses work

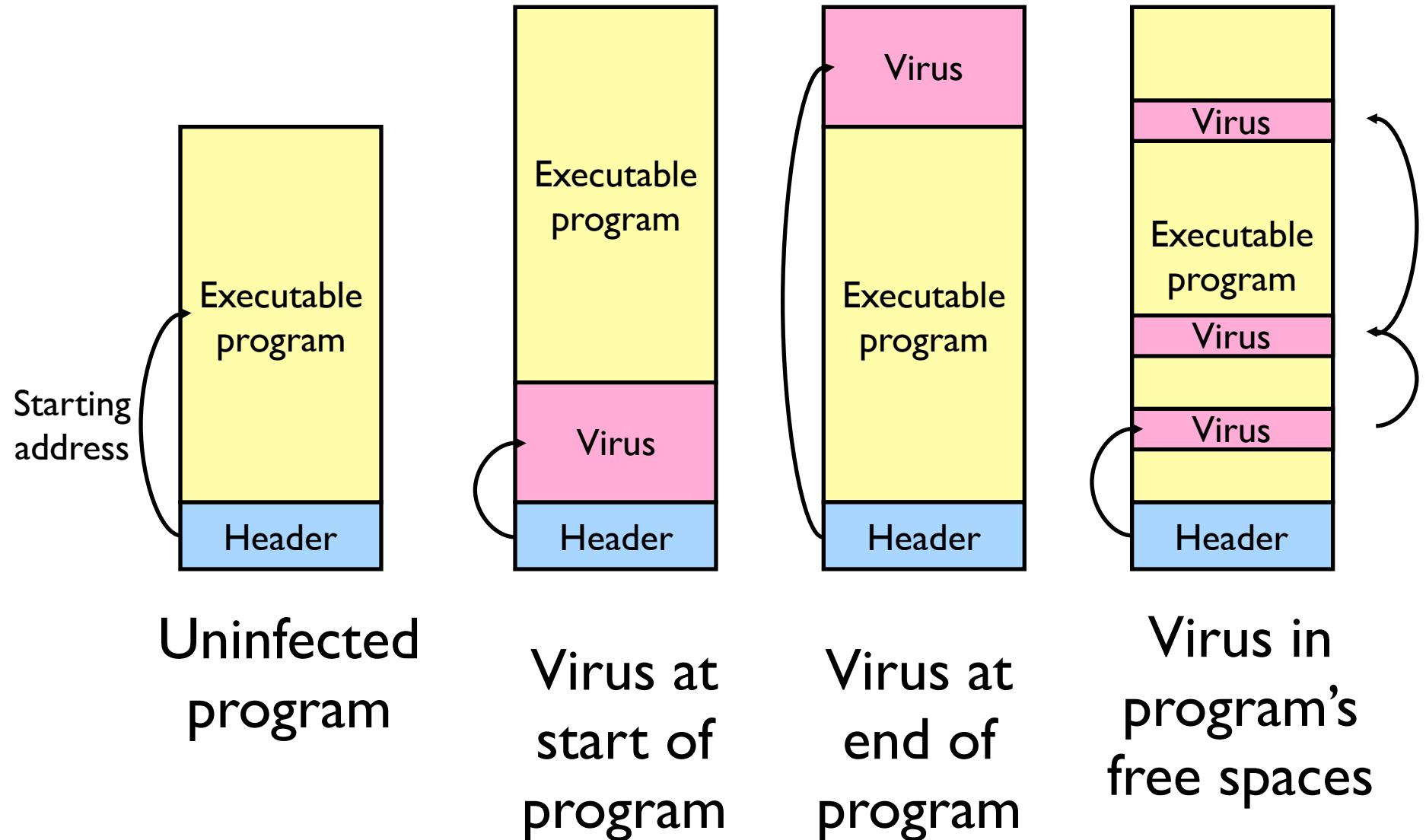
- ❖ Virus language
 - Assembly language: infects programs
 - “Macro” language: infects email and other documents
 - Runs when email reader / browser program opens message
 - Program “runs” virus (as message attachment) automatically
- ❖ Inserted into another program
 - Use tool called a “dropper”
 - May also infect system code (boot block, etc.)
- ❖ Virus dormant until program executed
 - Then infects other programs

How viruses find executable files

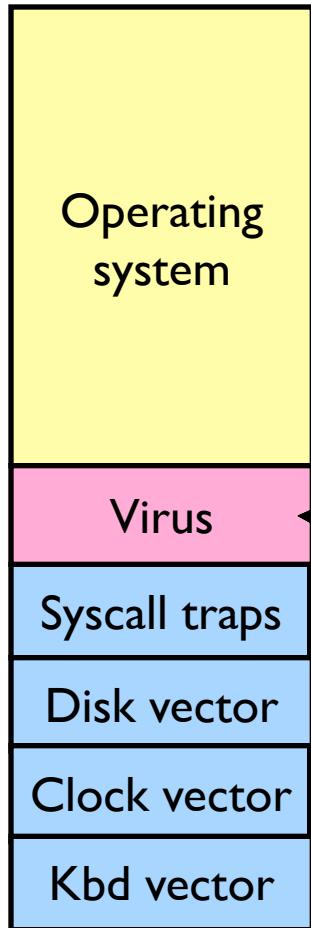
- ❖ Recursive procedure that finds executable files on a UNIX system
- ❖ Virus can infect some or all of the files it finds
 - Infect all: possibly wider spread
 - Infect some: harder to find?

```
#include <sys/types.h>          /* standard POSIX headers */  
#include <sys/stat.h>  
#include <dirent.h>  
#include <fcntl.h>  
#include <unistd.h>  
struct stat sbuf;  
  
search(char *dir_name)           /* for lstat call to see if file is sym link */  
{  
    DIR *dirp;                  /* recursively search for executables */  
    struct dirent *dp;          /* pointer to an open directory stream */  
    /* pointer to a directory entry */  
  
    dirp = opendir(dir_name);   /* open this directory */  
    if (dirp == NULL) return;   /* dir could not be opened; forget it */  
    while (TRUE) {  
        dp = readdir(dirp);    /* read next directory entry */  
        if (dp == NULL) {       /* NULL means we are done */  
            chdir("..");      /* go back to parent directory */  
            break;              /* exit loop */  
        }  
        if (dp->d_name[0] == '.') continue; /* skip the . and .. directories */  
        lstat(dp->d_name, &sbuf); /* is entry a symbolic link? */  
        if (S_ISLNK(sbuf.st_mode)) continue; /* skip symbolic links */  
        if (chdir(dp->d_name) == 0) { /* if chdir succeeds, it must be a dir */  
            search(".");         /* yes, enter and search it */  
        } else {                /* no (file), infect it */  
            if (access(dp->d_name,X_OK) == 0) /* if executable, infect it */  
                infect(dp->d_name);  
        }  
    closedir(dirp);           /* dir processed; close and return */  
}
```

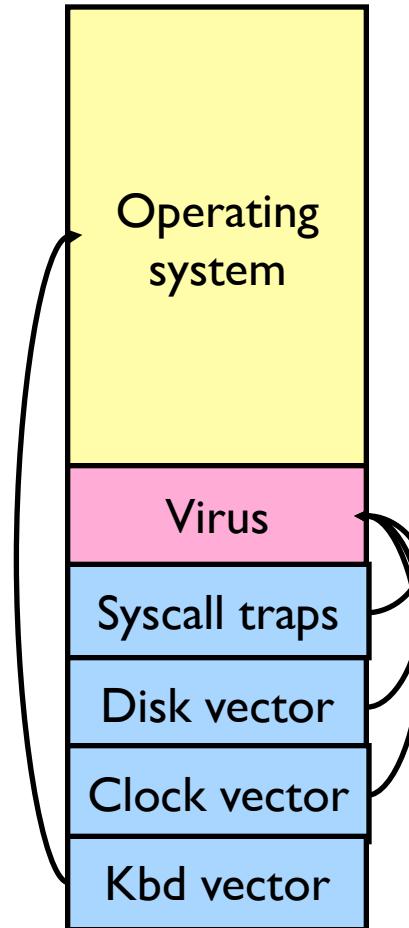
Where viruses live in the program



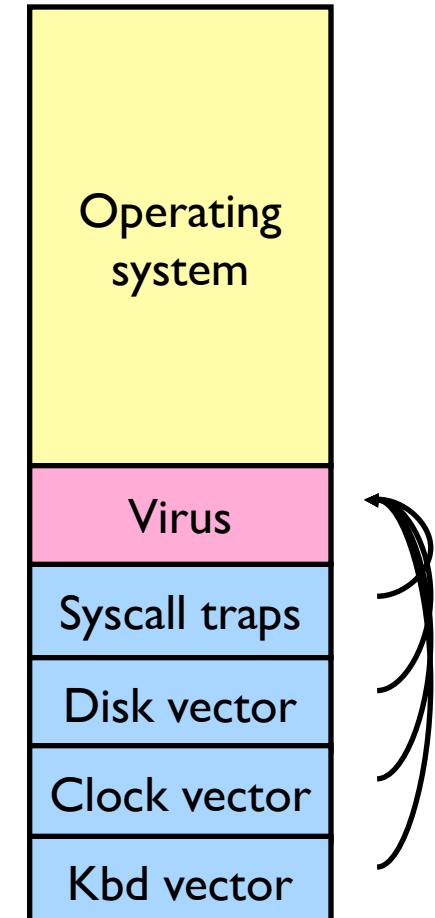
Viruses infecting the operating system



Virus has captured
interrupt & trap vectors



OS retakes
keyboard vector



Virus notices,
recaptures keyboard

How do viruses spread?

- ❖ Virus placed where likely to be copied
 - Popular download site
 - Photo site
- ❖ When copied
 - Infects programs on hard drive, thumb drive (even CD!)
 - May try to spread over LAN or WAN
- ❖ Attach to innocent looking email
 - When it runs, use mailing list to replicate
 - May mutate slightly so recipients don't get suspicious

Worms vs. viruses

- ❖ Viruses require other programs to run
- ❖ Worms are self-running (separate process)
- ❖ The 1988 Internet Worm
 - Consisted of two programs
 - Bootstrap to upload worm
 - The worm itself
 - Exploited bugs in sendmail and finger
 - Worm first hid its existence
 - Next replicated itself on new machines
 - Brought the Internet (1988 version) to a screeching halt

Spyware

- ❖ Type of trojan horse
 - Loaded onto PC without owner's knowledge
- ❖ Goals
 - Hides (so user can't find it)
 - Collects data about the user
 - Communicates collected data to spyware "master"
 - Tries to prevent itself from being removed
- ❖ Three main types
 - Marketing
 - Surveillance (companies often spy on employees!)
 - Turn computer into zombie
- ❖ May spread like a trojan horse

Rootkits

- ❖ Set of code and files that works hard to hide its existence
- ❖ Types of rootkits
 - Firmware (hides in BIOS)
 - Hypervisor (hides in VMM)
 - Kernel rootkit
 - Library rootkit (i.e., in libc)
 - Application rootkit
- ❖ Difficult to detect, especially in kernel or below!
 - Rootkit may return “fake” answers to hide its existence!
 - Rootkit uses resources: detect that usage
 - Rootkit may take longer to run: detect timing changes
- ❖ Sony included a rootkit on their music CDs to defeat attempts to copy the CD
 - The public was not amused...

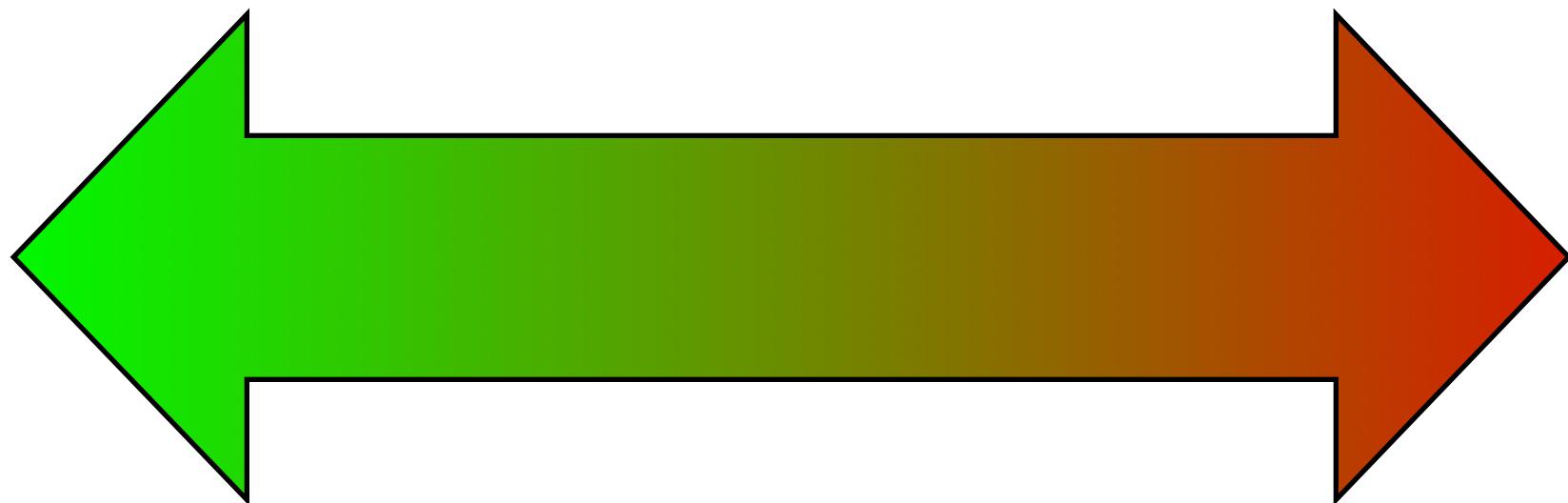
Defenses

- ✿ Defend against attacks using “defense in depth”
 - Multiple layers of security
 - Breaking a single layer doesn’t compromise the system
 - Layers need to be diverse!
- ✿ Many different types of defenses
 - Firewalls
 - Virus defenses
 - Code signing
 - Jails
 - Encapsulation of mobile code
 - Model-based intrusion detection

Design principles for security

- ❖ System design should be public
- ❖ Default should be no access
- ❖ Check for current authority
- ❖ Give each process least privilege possible
- ❖ Protection mechanism should be
 - Simple
 - Uniform
 - In the lowest layers of system
- ❖ Scheme should be psychologically acceptable
- ❖ Biggest thing: **keep it simple!**

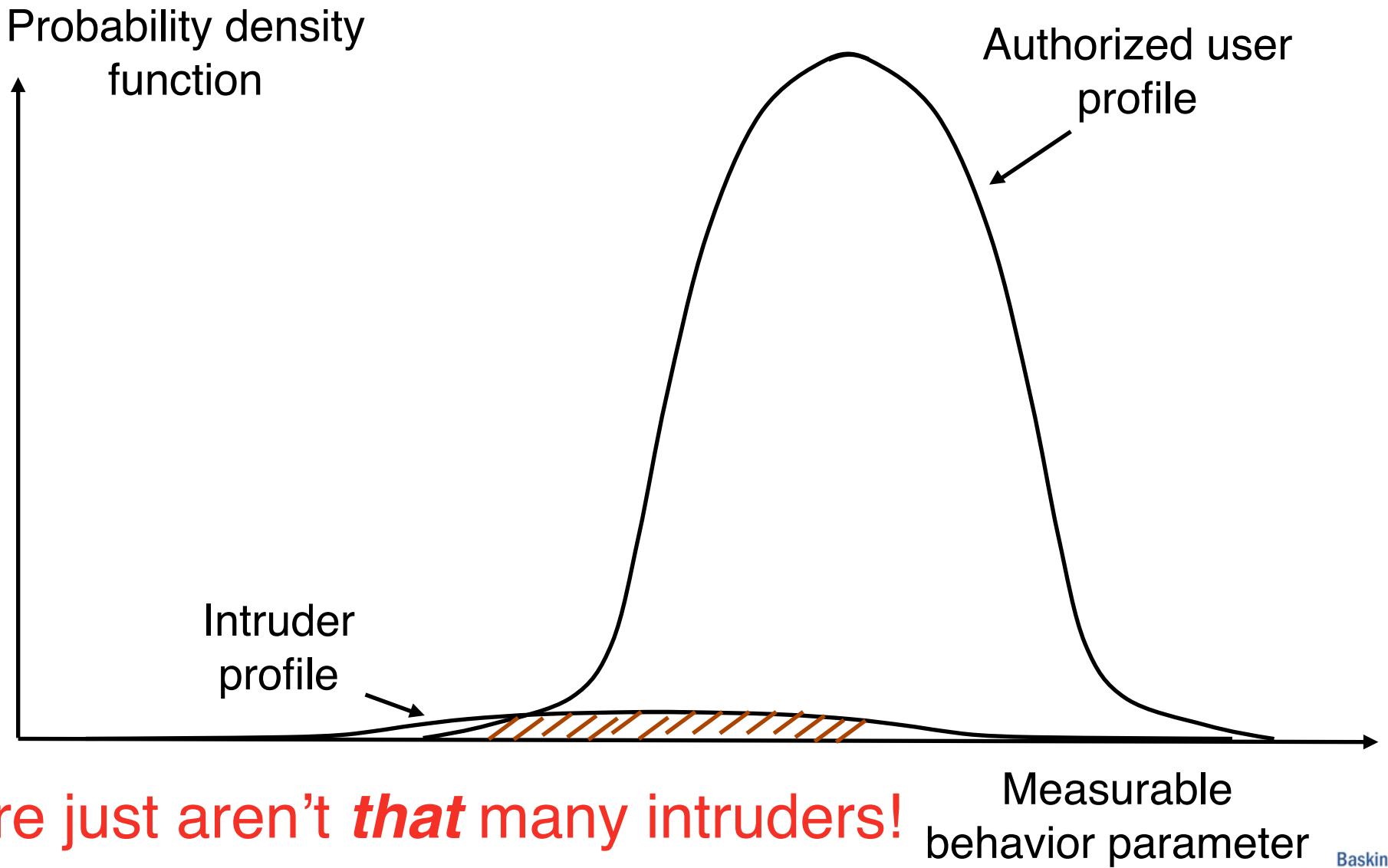
Range of user behavior



Obviously Normal

Obviously Malicious

Users & intruders



Problems with false positives

- ❖ Doctor invents a new, inexpensive test for a deadly disease that is 99% accurate
- ❖ Assume 1 in 2000 people have deadly disease (but don't know it yet)
- ❖ Should everyone get the test?
 - 2000 people tested
 - Expect $.99 + (1999 \times .01)$ positives
 - 21 people will be told they have disease
- ❖ If you test positive, there is about a 5% chance you have the disease
 - Higher than 1/2000, but not that high!

Firewalls: detecting and stopping intruders

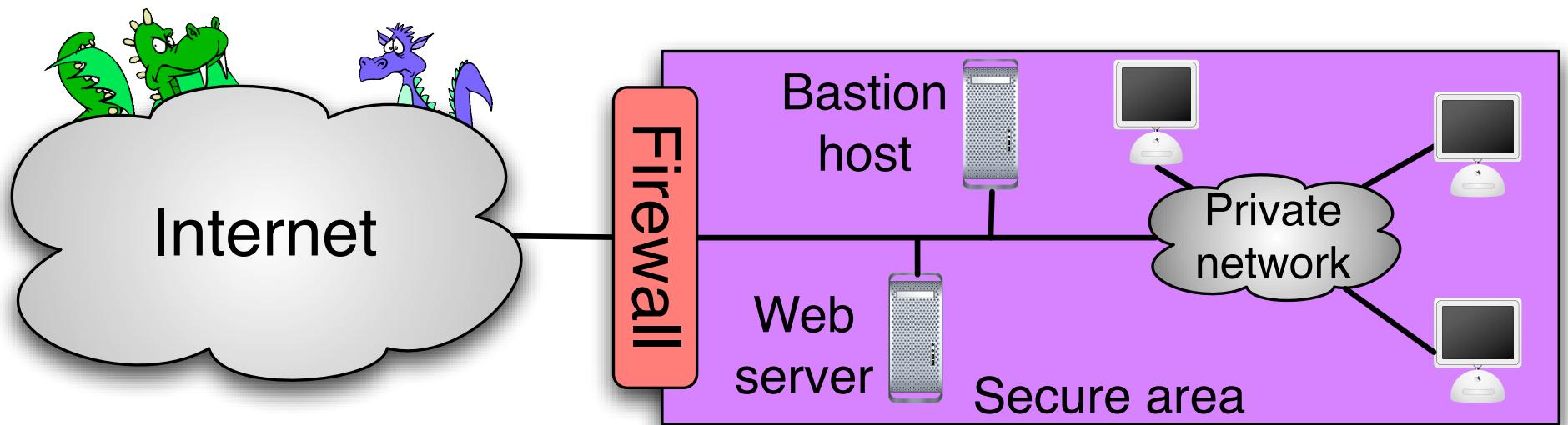
- ♦ Intrusion detection & prevention is hard!
 - Difficult to do well
 - Difficult to even detect an intrusion, particularly when it affects multiple computers
- ♦ One solution: use a firewall
- ♦ Firewall goals
 - All traffic between internal network and external network (Internet) must pass through the firewall
 - Only “authorized” traffic is allowed to pass (more on this in a bit)
 - Firewall itself is immune to penetration
 - This can be easier for a firewall because it doesn’t allow general-purpose logins
 - May not have a full operating system...

Guidelines for using firewalls

- ✿ Important: a single line of defense is not enough!
 - It's difficult to get through a firewall, but not impossible
 - Multiple lines of defense can slow down an attacker
 - Allow him to be detected
 - Allow the network to be cut off before there's further damage
- ✿ Configure multiple lines of defense so that they're more difficult than just one
- ✿ Protect more important resources with more defenses
- ✿ Use diverse types of firewalls and bastion hosts (systems that provide “external” services)!
 - Three firewalls of the same model may be no harder to compromise than one firewall
- ✿ Disallow logins, particularly from the outside world
 - It's more difficult to get access to a serial line than to a Web browser interface
 - Single-purpose OS is more difficult to get into

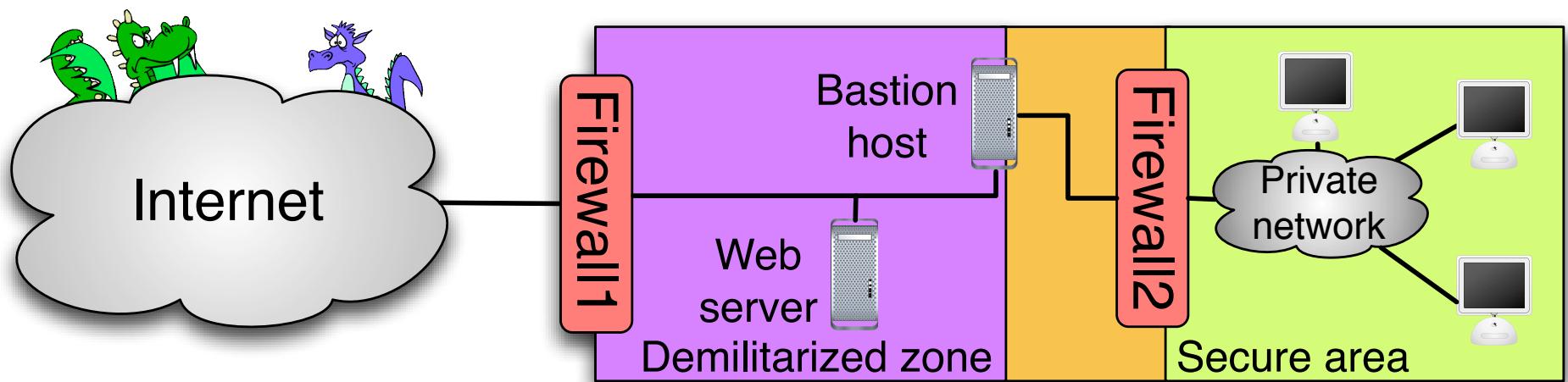
Basic firewall configuration

- ❖ Traffic controlled by firewall
- ❖ Drawbacks
 - Firewall has to do a lot of work
 - Bastion host can be bypassed
 - Compromises hard to deal with: isolation is difficult



Two firewalls, three zones

- ❖ More complex configuration for more security
- ❖ Multiple zones
 - DMZ
 - Area between bastion host and private network
 - Allows for more filtering
 - Must now break into two firewalls as well as bastion host
- ❖ Private network doesn't know (directly) about Internet
- ❖ Internet doesn't know about private network

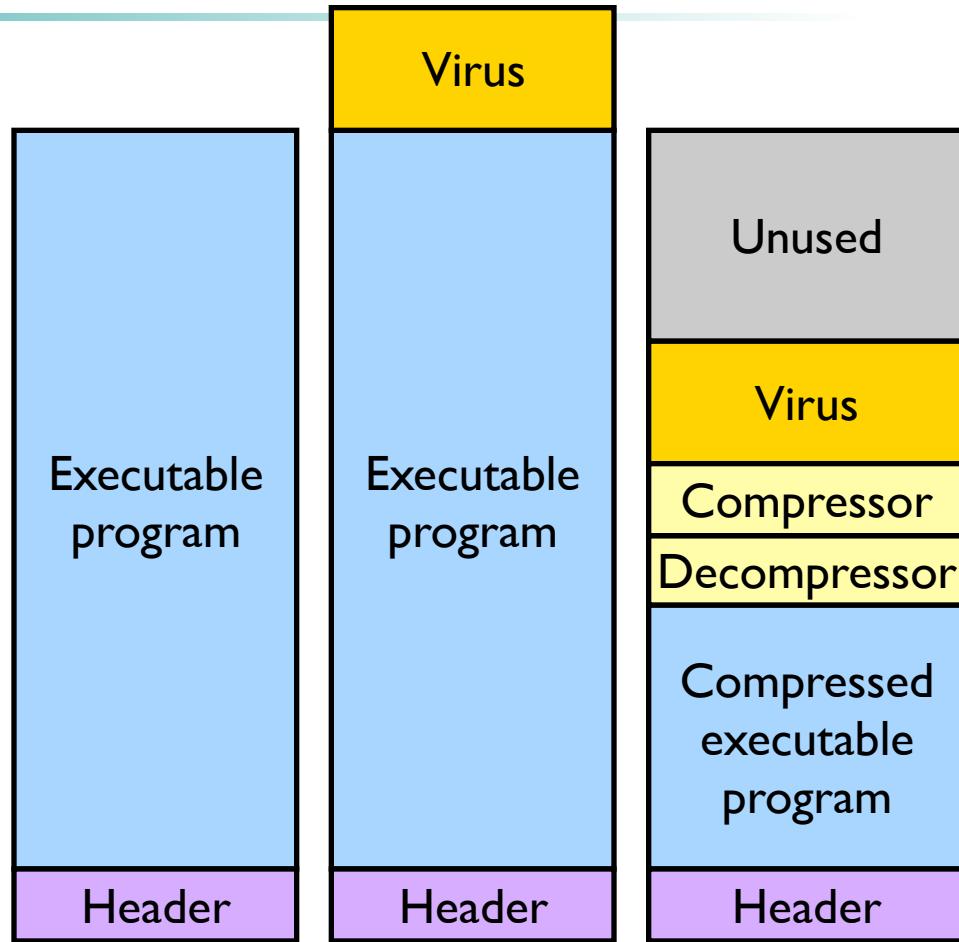


Stopping viruses

- ❖ Battle against viruses is somewhat of an arms race
 - Virus scanners & detection get better
 - Viruses get better at avoiding detection
- ❖ Several major approaches to detect and handle viruses
 - Find particular code that makes up a virus
 - Check to ensure that programs aren't infected
 - Check for aberrant behavior
- ❖ Good idea to practice “safe computing”...

Hiding a virus in a file

- ❖ Start with an uninfected program
- ❖ Add the virus to the end of the program
 - Problem: file size changes
 - Solution: compression
- ❖ Compressed infected program
 - Decompressor: for running executable
 - Compressor: for compressing newly infected binaries
 - Lots of free space (if needed)
- ❖ Problem (for virus writer): virus easy to recognize

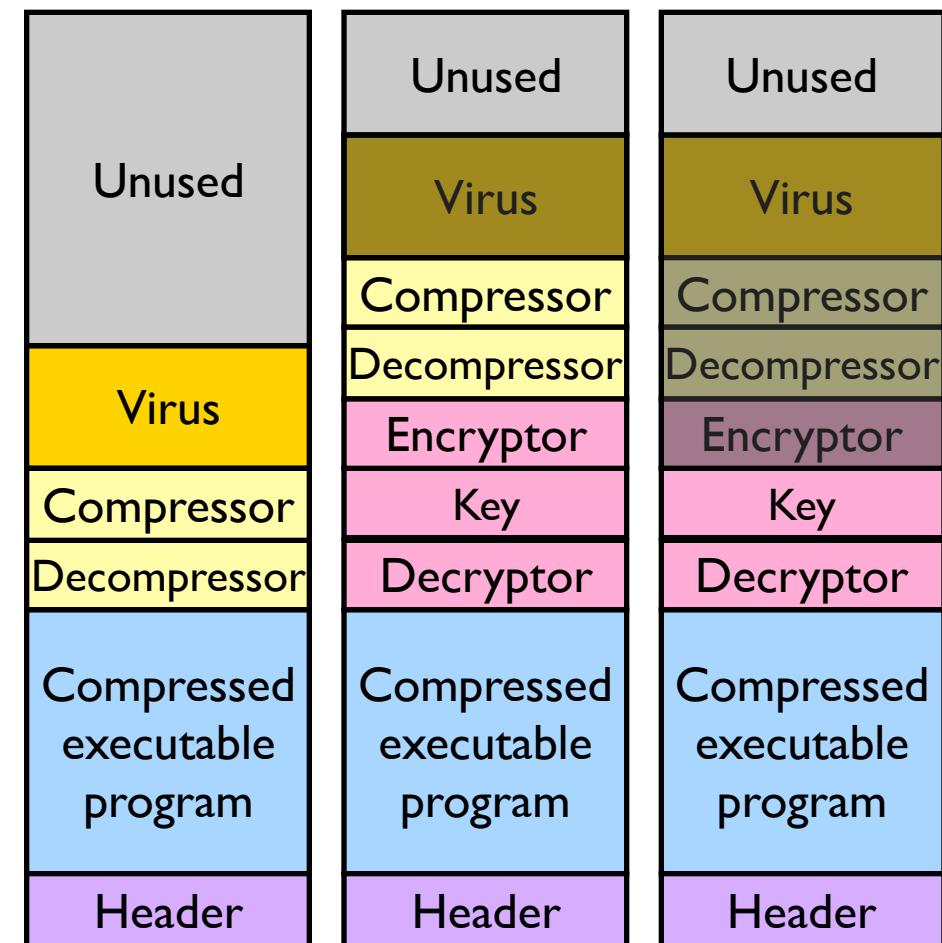


Virus scanning

- ❖ Look through programs for sequences of code that are “characteristic” of viruses
 - Need to have a library of virus definitions
 - Need to allow for (at least) minor variations
- ❖ Several potential problems with this
 - Need the virus definitions before a new virus can be found
 - Infections can happen very quickly!
 - Viruses may not have characteristic signatures
 - Viruses may change their signatures over time...

Using encryption to hide a virus

- ♦ Hide virus by encrypting it
 - Vary the key in each file
 - Virus “code” varies in each infected file
 - Problem: lots of common code still in the clear
 - Compress / decompress
 - Encrypt / decrypt
- ♦ Even better: leave only decryptor and key in the clear
 - Less constant per virus
 - Use polymorphic code (more in a bit) to hide even this



Polymorphic viruses

- ❖ All of these code sequences do the same thing
- ❖ All of them are very different in machine code
- ❖ Use “snippets” combined in random ways to hide code

MOV A,R1
ADD B,R1
ADD C,R1
SUB #4,R1
MOV R1,X

(a)

MOV A,R1
NOP
ADD B,R1
NOP
ADD C,R1
NOP
SUB #4,R1
NOP
MOV R1,X

(b)

MOV A,R1
ADD #0,R1
ADD B,R1
OR R1,R1
ADD C,R1
SHL #0,R1
SUB #4,R1
JMP .+1
MOV R1,X

(c)

MOV A,R1
OR R1,R1
ADD B,R1
MOV R1,R5
ADD C,R1
SHL R1,0
SUB #4,R1
ADD R5,R5
MOV R1,X
MOV R5,Y

(d)

MOV A,R1
TST R1
ADD C,R1
MOV R1,R5
ADD B,R1
CMP R2,R5
SUB #4,R1
JMP .+1
MOV R1,X
MOV R5,Y

(e)

Other approaches to detecting viruses

- ❖ Integrity checking
 - Record the cryptographic checksum of uninfected programs
 - Periodically recalculate checksum and verify against saved values
 - If there's any change, a file may be infected
 - Issue: checksum file (and checker) need to be clean!
- ❖ Behavioral checking
 - Antivirus program checks to ensure that applications don't make "bad" system calls
 - Notify user if any program tries to do something
 - User needs to decide if it's legitimate or not

Avoiding viruses in the first place

- ❖ Choose an OS that has high virus resistance
 - Built-in features that make viruses harder to write
- ❖ Use only shrink-wrapped software or shareware from *trusted* sources
 - Other software may look useful, but is it infected?
 - Viruses & spyware often spread this way!
- ❖ Deploy a good anti-virus program and update it regularly
- ❖ Don't download random email attachments or visit random Web pages on unfamiliar sites
- ❖ Make backups!
 - Keep backups on external (non-connected) media
 - Good idea for other reasons, too...

Code signing

- ♦ Goal: only run software from reliable vendors
- ♦ How can the system tell that the code is from a reliable vendor (and unmodified)?
- ♦ Solution: *code signing*
 - Software designer hashes the program
 - Designer's private key is used to encrypt the hash, generating a signature
 - User can decrypt the signature and compare the result to the locally-computed hash
 - If they match, code is safe
- ♦ Requires signature infrastructure similar to that needed for secure Web pages...

Jails and sandboxes

- ❖ Goal: isolate software from the rest of the system
- ❖ Mechanisms
 - Jails
 - Sandboxes
- ❖ Jail: all system calls go through a jailer
 - Jailer decides whether or not to allow the call
 - Example: jails often have a “root directory” below the system-level root
 - Attempts to escape this new root directory are foiled
- ❖ Sandbox: confine application to a fixed area of memory
 - More on this in a bit...

Rule (model)-based intrusion detection

- ❖ Identify intruders by what they're trying to do
- ❖ Model shows “normal” behavior
 - Generated by compiler
 - Generated by operating system from observing normal code
- ❖ Deviation from model indicates an attack
 - Some deviation is normal!
 - How much deviation is too much?

Honeypots

- ♦ To keep the system safe, set up resources that an attacker can (relatively easily) get a hold of
- ♦ Make sure that these resources can't lead to further breakins
- ♦ Example: fake user & shell
 - Set up fake user names & passwords with rewritten shell
 - Ensure that these user names are easier to break into
 - Simpler passwords
 - Remote shell
 - Make fake data available to shell, perhaps using chroot()
 - Install fake programs such as ls and telnet/ssh to track what the intruder is doing
 - Programs appear to work, but provide false data
- ♦ Keep the intruder on the system as long as possible to trace where he's coming from!

Mobile code

- ❖ Goal: run (untrusted) code on my machine
- ❖ Problem: how can untrusted code be prevented from damaging my resources?
- ❖ One solution: *sandboxing*
 - Memory divided into sandboxes
 - Accesses may not cross sandbox boundaries
 - Sensitive system calls not in the sandbox
- ❖ Another solution: interpreted code
 - Run the interpreter rather than the untrusted code
 - Interpreter doesn't allow unsafe operations
 - Run code in a virtual machine (VMware?)
- ❖ Third solution: signed code

Security in Java

- ♦ Java is a type safe language
 - Compiler rejects attempts to misuse variable
- ♦ No “real” pointers
 - Can’t simply create a pointer and dereference it as in C
- ♦ Checks include ...
 - Attempts to forge pointers
 - Violation of access restrictions on private class members
 - Misuse of variables by type
 - Generation of stack over/underflows
 - Illegal conversion of variables to another type
- ♦ Applets can have specific operations restricted
 - Example: don’t allow untrusted code access to the whole file system