Computer Security

Chapter Seven

Overview

- Three important computer security components:
  - User authentication – determine the identity of an individual accessing the system
  - Access control policies – stipulate what actions a given user is allowed to perform on the system
  - Access control mechanisms – enforce the system's access control policy

Policy

- In general, to protect an object, an operating system must define:
  - A policy, which states what controls are to be enforced for that object, and
  - A mechanism, which implements the policy.

Reference Monitor

- **Reference monitor** controls access to system objects
- Reference monitor must:
  - Operate correctly
  - Always be invoked
  - Be tamper-proof
- Decisions on whether or not to allow an action are based on the **identity** of the user performing the action

Authorization

- **Authorization** entails determining whether or not the protection policy permits a given user to perform a given action
- Many operating systems base authorization decisions on a user’s unique user identifier (or *uid*):
  - User is authenticated during log on and given an appropriate uid
    - Must enter valid username and password
  - The uid is used to determine which actions are authorized

User Authentication

- Three basic approaches:
  - **Knowledge-based –**something they know
    - Example: passwords
  - **Token-based –**through something they possess
    - Example: passport
  - **Biometric –**unique physiological characteristic
    - Example: fingerprint

Passwords

- Widely-used
- Advantages:
  - Easy to use
  - Understood by most users
  - Requires no special equipment
- Disadvantages:
  - Users tend to choose easy passwords
  - Many password-cracking tools available

Using Passwords

- User enters username and password
- The operating system consults its table of passwords:

Username	Uid	Password
Alice	12	dumptruck
Bob	7	baseball

- Match = user is assigned the corresponding uid
- Problem: the table of passwords must be protected

Using Passwords and One-Way Functions

- User's password not stored in the table
- One-way password hash, *h*(password), stored in the table
  - *h*(dumptruck) = JFNXPEMD
  - *h*(baseball) = WSAWFFVI

Username	Uid	Hash
Alice	12	JFNXPEMD
Bob	7	WSAWFFVI

Using Passwords and One-Way Functions (cont)

- User enters username and password
- The operating system hashes the password
- The operating system compares the result to the entry in the table
- Match = user is assigned the corresponding uid
- Advantage: password table does not have to be protected
- Disadvantage: dictionary attack

A Dictionary Attack

- An attacker can compile a **dictionary** of several thousand common words and compute the hash for each one:

Password	Hash
Baseball	WSAWFFVI
Basketball	BFQLSZAY
Football	ORCVVGTS
. . .	. . .

- Look for matches between the dictionary and the password table
  - Example: WSAWFFVI tells us Bob's password is baseball

Dictionary Attacks (cont)

- Dictionary attacks are a serious problem:
  - Costs an intruder very little to send tens of thousands of common words through the one-way function and check for matches
- Solution #1: don't allow users to select their own passwords
  - System generates a random password for each user
  - Drawback:
    - Many people find system-assigned passwords hard to remember and write them down
      - Example: L8f#n!.5rH'

Combating Dictionary Attacks

- Solution #2: password checking
  - Allow users to choose their own passwords
  - Do not allow them to use passwords that are in a common dictionary
- Solution #3: salt the password table
  - A **salt** is a random string that is concatenated with a password before sending it through the one-way hash function
    - Random salt value chosen by system
      - Example: plre
    - Password chosen by user
      - Example: baseball

Salting the Password Table

- Password table contains:
  - Salt value = plre
  - h(password+salt) = h(baseballplre) = FSXMXFNB

Username	Uid	Salt	Hash
Alice	12	DCFV	IGHERVCL
Bob	7	PLRE	FSXMXFNB

Salting the Password Table (cont)

- User enters username and password
- The operating system combines the password and the salt and hashes the result
- The operating system compares the result to the entry in the table
- Match = user is assigned the corresponding uid
- Advantages:
  - Password table does not have to be protected
  - Dictionary attacks much harder

A Dictionary Attack

- Attacker must now expand the dictionary to contain every possible salt with each possible password:
  - baseballaaaa
  - baseballaaab
  - baseballaaac

    ….
  - baseballaaaz
  - baseballaaba
  - baseballaabb

    ….
- 26^4 (about half a million) times more work to check each word in the dictionary (for 4-letter salts)

Access Control Policies

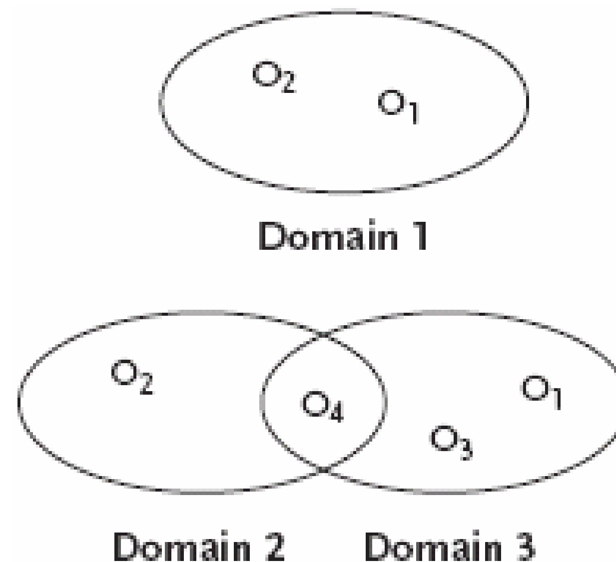
- Once a user has logged in the system must decide which actions he can and cannot perform
  - Examples:
    - Bob may be allowed to read files that Alice cannot
    - Alice may be permitted to use a printer that Bob cannot
- In general, we view the system as a collection of:
  - Subjects (users)
  - Objects (resources)
- An **access control policy** specifies how each subject can use each object

The Access Control Matrix

- Forms the basis of protection in many real operating systems
  - Resources to be protected are called **objects**
  - Every object is within one or more protection **domain(s)**
  - A domain specifies what operations are permitted on the objects it contains
  - Authorization to perform an operation on an object in a domain is called an **access right**
  - Suggested by Butler Lampson

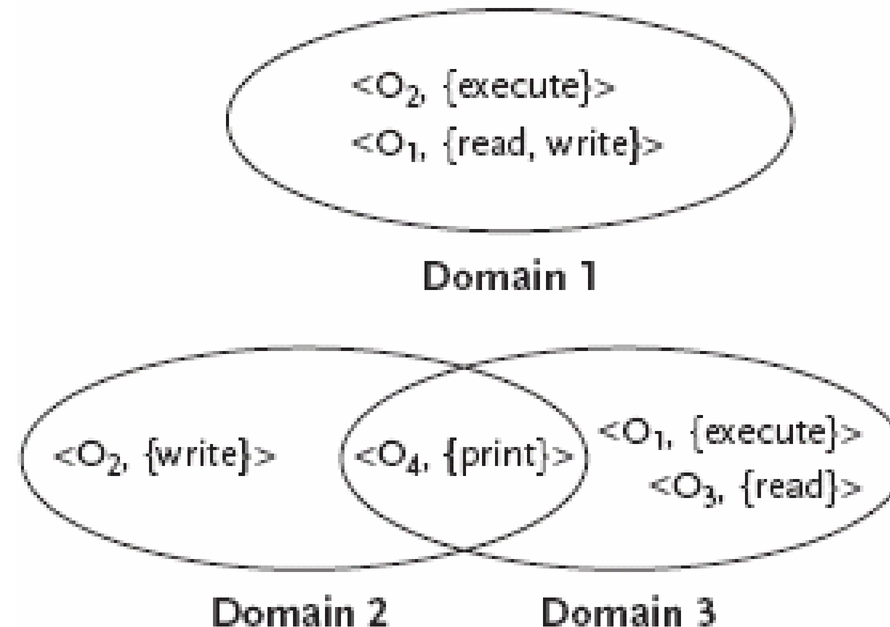
Protection Domains - Example

- Assume:
  - All students are in domain 1
  - All faculty members are in domain 2
  - All system administrators are in domain 3



Protection Domains – Example (cont)

- Students can execute O_2 and read and write O_1
- Faculty can write O_2 and print O_4
- Sys admins can execute O_1, read O_3, and print O_4



Protection Domains (cont)

- A user can only be in one protection domain at any given time
  - Static: a user always operates in the same domain
    - Simple
    - Inflexible
  - Dynamic: a user can switch from one domain to another
    - Complex
    - More flexible
- The domain in which a user is operating determines what actions are and are not permitted

Lampson's Access Control Matrix

- Represent the protection domains and access rights using a matrix:
  - Rows represent the domains
  - Columns correspond to the objects
  - Matrix entries specify the access rights to an object in the corresponding domain

	Object 1	Object 2	Object 3	Object 4
Domain 1	{read, write}	{execute}		
Domain 2		{write}		{print}
Domain 3	{execute}		{read}	{print}

Access Control Policy

- The matrix specifies an access control **policy** for the system:
  - Stipulates what actions a given user is allowed to perform on the system
  - Examples:
    - A student (domain 1) is allowed to read object 1
    - A faculty member (domain 2) is not allowed to read object 1
    - A faculty member (domain 2) is allowed to write to object 2

	Object 1	Object 2	Object 3	Object 4
Domain 1	{read, write}	{execute}		
Domain 2		{write}		{print}
Domain 3	{execute}		{read}	{print}

Access Control Mechanisms

- A **mechanism** is required to enforce the access control policy
- Mechanism #1: store the matrix in (protected) memory
  - The matrix is likely to be large and sparse so store only non-empty entries
  - One ordered triple that represents the domain, object, and access rights corresponding to each non-empty cell in the matrix:

    (Domain 1, Object 1, {read, write})

    (Domain 1, Object 2, {execute})

    (Domain 2, Object 2, {write})

    (Domain 2, Object 4, {print})

    (Domain 3, Object 1, {execute})

    (Domain 3, Object 3, {read})

    (Domain 3, Object 4, {print})

Access Control Mechanisms (cont)

- An attempt by a user in Domain i to perform operation O on Object j
  - The operating system consults the list of triples
    - Allow: if there is a triple (i, j, R) where $O \in R$, the access rights
    - Deny: otherwise
  - Examples:
    - Allow: attempt to write to Object 2 by a user operating in Domain 2
    - Deny: attempt by a user in Domain 1 to write to Object 2

(Domain 1, Object 1, {read, write})

(Domain 1, Object 2, {execute})

(Domain 2, Object 2, {write})

…

Access Control Mechanisms (cont)

- Mechanism #1: store the matrix in (protected) memory
- Drawbacks:
  - The matrix must be protected against unauthorized modification
  - Searching may be slow (if there are many entries)
  - Cannot take advantage of special groupings of objects
    - Example: the system clock
      - Can be read by all users
      - Matrix requires an entry in every domain specifying that the clock is readable

Access Control Mechanisms (cont)

- Mechanism #2: represent the matrix as an access list
  - An **access list** enumerates, for each object, a set of (domain, access rights) pairs
  - Matrix:

	Object 1	Object 2	Object 3	Object 4
Domain 1	{read, write}	{execute}		
Domain 2		{write}		{print}
Domain 3	{execute}		{read}	{print}

  - Access list:

    Object 1: (<Domain 1, {read, write}>, <Domain 3, {execute}>)

    Object 2: (<Domain 1, {execute}>, <Domain 2, {write}>)

    Object 3: (<Domain 3, {read}>)

    Object 4: (<Domain 2, {print}>, <Domain 3, {print}>)

Access Control Mechanisms (cont)

- An attempt by a user in Domain *i* to perform operation *O* on Object *j*
  - The operating system to consult the entry in the access list for Object *j*
  - Object *j*'s list is searched for Domain *i*'s entry
    - Allow: if there is an access right for *O*
    - Deny: Otherwise
  - Access list:

    Object 1: (<Domain 1, {read, write}>, <Domain 3, {execute}>)

    Object 2: (<Domain 1, {execute}>, <Domain 2, {write}>)

    Object 3: (<Domain 3, {read}>)

    Object 4: (<Domain 2, {print}>, <Domain 3, {print}>)
  - Examples:
    - Allow: attempt to write to Object 2 by a user operating in Domain 2
    - Deny: attempt by a user in Domain 1 to write to Object 2

Access Control Mechanisms (cont)

- Access lists also allow a **default** set of rights to be specified for each object
- Example: make Object 2 readable in every domain:

  Object 2: (<Default, {read}>, <Domain 1, {execute}>, <Domain 2, {write}>)
- Advantages:
  - Can specify rights available in every domain in a single entry
    - Reduces the total amount of storage space required
    - Increases efficiency (for default rights)
- Disadvantages:
  - Must be stored in memory, protected, and searched

Access Control Mechanisms (cont)

- Mechanism #3: represent the matrix as capabilities
  - A **capability list** represents the matrix as a list of (object, rights) pairs for each domain
  - Matrix:

	Object 1	Object 2	Object 3	Object 4
Domain 1	{read, write}	{execute}		
Domain 2		{write}		{print}
Domain 3	{execute}		{read}	{print}

  - Capability list:

Domain 1: (<Object 1, {read, write}>, <Object 2, {execute}>)

Domain 2: (<Object 2, {write}>, <Object 4, {print}>)

Domain 3: (<Object 1, {execute}>, <Object 3, {read}>, <Object 4, {print}>)

Access Control Mechanisms (cont)

- Capability lists are **not** stored in memory by the operating system
- Users are given a copy of the capability list for the domain in which they are operating
- Each (object, rights) pair for a given domain is called a **capability**
- Example:
  - A user in Domain 2 would be given copies of the following two capabilities:
    - <Object 2, {write}>
    - <Object 4, {print}>

Access Control Mechanisms (cont)

- An attempt to perform some operation, *O*, on Object *j*
  - The capability for *j* is passed as one of the parameters of *O*
- Example: a user (in Domain 2) might request to write to Object 2
  - Pass her copy of the <Object 2, {write}> capability
  - The operating system verifies:
    - The capability is for Object 2
    - Writing is one of the access rights in the capability
  - Allow: if the capability is valid
  - Deny: otherwise

Access Control Mechanisms (cont)

- The operating system must ensure that users cannot:
  - Create their own capabilities
  - Alter the capabilities they are given
- Solution: operating system encrypt capabilities with a secret key before giving them to the users
- Capabilities also allow users to **share** their access rights with others:
  - If Alice has the right to read a file but Bob does not
  - Alice can give Bob a copy of her capability that will allow him to read the file

Access Control Mechanisms (cont)

- Mechanism #3: represent the matrix as capabilities
- Advantages:
  - OS need not store all the access-control information
    - Capabilities are given to users and stored by them
  - Access control decisions require no searching of a list
    - Capabilities need to be checked
  - Allow users to share their capabilities with others
- Disadvantages:
  - Capabilities must be protected (encrypted)
  - Sharing and revoking capabilities is complex

Other Access Control Policies

- The **HRU** model
  - Named for its inventors, Harrison, Ruzzo, Ullman
  - Describes a protection system using an access matrix
    - Rows: the subjects in the system (S_1, S_2, and S_3)
    - Columns: the objects in the system (S_1, S_2, S_3, O_1, O_2, and O_3)
    - Note: the subjects are considered objects in the system
  - A subject is allowed to perform an operation on an object only if it has an access right

The HRU Model

- The access matrix:

	S_1	S_2	S_3	O_1	O_2	O_3
S_1	Control				Owner Read	
S_2		Control		Owner Read Write	Read	Owner Execute
S_3			Control	Read	Read	Execute

- Examples:
  - S_1 should be allowed to read O_2
  - S_3 should not be allowed to write O_1

The HRU Model (cont)

- The HRU model allows the protection system to change:
  - Subjects and objects can be created or destroyed
  - Access rights can be entered or deleted from the matrix
- Example: S_3's right to read O_2 can be deleted:

	S_1	S_2	S_3	O_1	O_2	O_3
S_1	Control				Owner Read	
S_2		Control		Owner Read Write	Read	Owner Execute
S_3			Control	Read		Execute

The HRU Model (cont)

- The matrix can be changed by a well-defined set of commands that consist of:
  - A list of conditions that must be satisfied
  - A list of primitive operations that modify the matrix
    - Add/delete a subject to the matrix
    - Add/delete an object to the matrix
    - Add/delete a right to the matrix
- Example: the command used to delete S_3's right to read O_2
  - If a subject is the owner of an object
  - The subject can perform the delete operation on a right in the matrix

The HRU Model (cont)

- A right, *r*, is said to **leak** if:
  - The execution of some set of commands causes *r* to be entered into a cell that did not previously contain *r*
- A state where *r* has leaked is called **unsafe**
- If, starting from an initial state unsafe states are unreachable:
  - The matrix is said to be **safe** for *r,* since *r* cannot be leaked

The HRU Model (cont)

- The HRU model was designed to be general enough to represent most real protection systems
- Main result: Harrison, Ruzzo, Ullman proved that:
  - Determining whether or not unsafe states are reachable from a given initial state is not possible for an arbitrary set of commands
  - Safety is decidable in some restricted cases:
    - No command contains an operation that add subjects or objects
    - **Mono-operational** systems: every command performs only a single primitive operation

The HRU Model (cont)

- The HRU model demonstrates that for most useful access control policies:
  - Substantial restrictions must be placed on the policy to facilitate analysis
  - Even with restrictions analysis is likely to be difficult
  - If the policy is proven safe, we still need to show that the system properly implements the policy

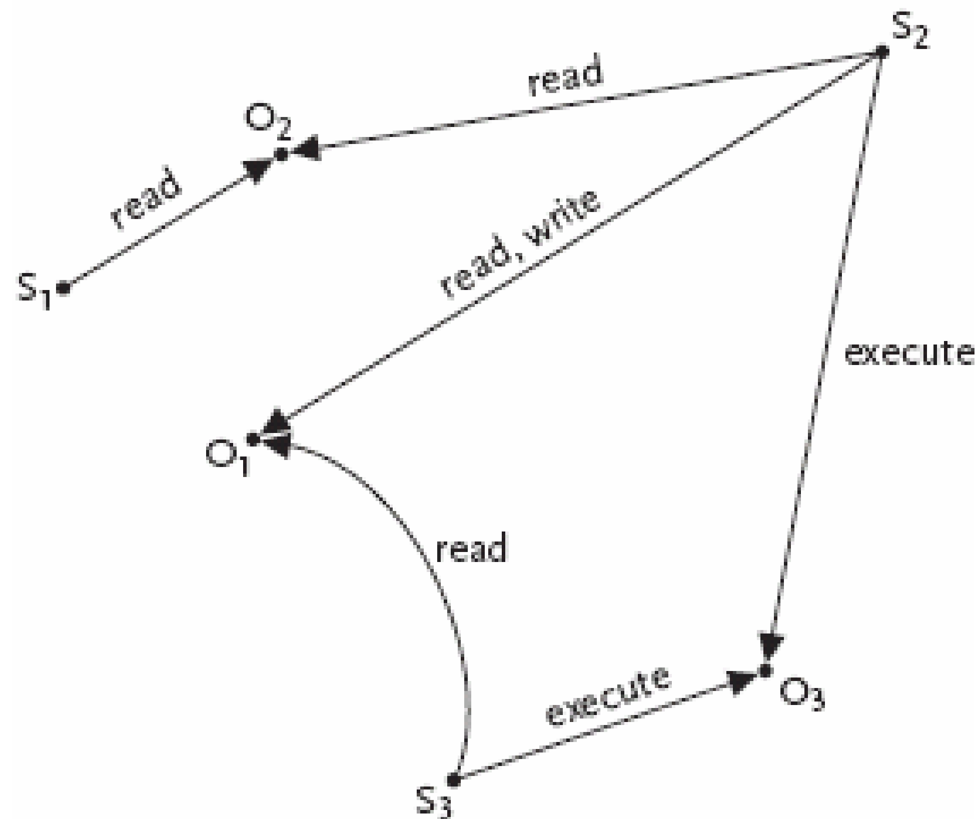
Other Access Control Policies (cont)

- The **take-grant** model:
  - More restricted than the HRU model
    - Not powerful enough to represent an arbitrary protection policy
  - Designed to be both useful and efficiently analyzable
  - An unlimited number of subjects and objects can be created
  - Four primitive operations:
    - Take
    - Grant
    - Create
    - Revoke

The Take-Grant Model

- A policy is not represented as a matrix
- A policy is represented as a directed graph
  - Nodes represent subjects and objects
    - Unlike the HRU model, subjects are not themselves objects
  - Edges represent access rights
- Safety question:
  - Can a subject gain a given access right to a given object?

The Take-Grant Model (cont)



The Take-Grant Model (cont)

- The **create** operation
  - Allows a subject to add a new object node to which the subject then has a certain set of access rights
  - Example: *S* executes *create*(*O*,{*read*,*write*}):

- The **revoke** operation
  - Allows a subject to delete one or more of its access rights to an object
  - Example: *S* executes *revoke*(*O*,{*write*}):

The Take-Grant Model (cont)

- The **take** operation
  - Allows a subject to acquire one or more of the access rights possessed by another subject or object
  - Example: *S1* takes *S2*'s read right:

- The **grant** operation
  - Allows a subject to confer one or more of its access rights to another subject or object
  - Example: *S1* grants *S2* a read right:

The Take-Grant Model (cont)

- More restricted than the HRU model
  - Not powerful enough to represent an arbitrary protection policy
- Designed to be both useful and efficiently analyzable
  - There is an efficient algorithm for determining whether or not a particular subject can acquire a specific right to an object
- Work has been done to define useful protection systems based on the take-grant model

Discretionary Vs. Mandatory Access Control

- All the access control policies and mechanisms discussed so far have been **discretionary**
  - The owner of an object fully controls what users have what access rights
- **Mandatory** access control policies and mechanisms
  - Control over what users have what access rights resides with the system and not the object's owner
- Examples: Bob creates a file
  - Discretionary: Bob can give Alice read access to the file if he wishes
  - Mandatory: The system may deny Alice read access to Bob's file regardless of Bob's wishes

Information Flow Policies

- An important class of mandatory access control policies are **information flow policies**
  - Control both direct and indirect paths through which information can flow
- Example:
  - Alice does not have permission to read a file but Bob does
  - Alice convince Bob to read the file and transmit its contents to her
- Result: Alice has gained unauthorized access to the file's contents
  - Discretionary access-control policies cannot prevent this breach
  - Information flow policies can prevent this breach

The Bell-LaPadula Model

- An information flow security policy
- Distinguishes between different levels of sensitivity for information and users
  - Information: Public records are not sensitive
  - Users: Everybody should have access to public records
  - Information: The tax returns of individuals are somewhat sensitive
  - Users: Only employees of the Internal Revenue Service should have access to tax returns
  - Information: Nuclear secrets are extremely sensitive
  - Users: Only the most trusted nuclear scientists should have access to nuclear secrets

The Bell-LaPadula Model (cont)

- Every user and every piece of information can be assigned a **level** from a partially ordered set
- Example: levels used by the U.S. military
  - The set has four elements:
    - L = {unclassified, classified, secret, top secret}
  - The $\leq$ relational operator is used to specify a partial ordering on the set:
    - unclassified $\leq$ classified $\leq$ secret $\leq$ top secret
  - This is a **partial ordering** since the $\leq$ relation is:
    - Reflexive ($L1 \leq L1$)
    - Antisymmetric (if $L1 < L2$ and $L2 < L1$, then $L1 = L2$)
    - Transitive (if $L1 < L2$ and $L2 < L3$, then $L1 < L3$)

The Bell-LaPadula Model (cont)

- Every subject and object in the system is assigned a security level from *L*
- Example:
  - Alice might be assigned the level secret
  - Bob might be unclassified
  - Carol might be classified
  - The file memo1 might be classified
  - The file memo2 might be top secret

The Bell-LaPadula Model (cont)

- The information flow security policy:
  - The **simple security property**: a subject should not be able to read an object at a higher level
    - A subject, *S*, may read an object, *O*, only if $O < S$
  - The **star property** (written "*-property"): a subject should not be able to write to an object at a lower level
    - A subject, *S*, may write to an object, *O*, only if $S < O$

The Bell-LaPadula Model (cont)

- Example:
  - Alice's level is secret, Bob's level is unclassified, Carol's level is classified
  - Memo1 is classified, and memo2 is top secret
- The simple security property specifies that:
  - Memo2 should not be read by Alice, Bob, or Carol
    - Memo2's level (top secret) is higher than theirs (secret, unclassified, and classified, respectively)
  - Bob is not allowed to read memo1 (classified), but both Alice and Carol are allowed to read it

The Bell-LaPadula Model (cont)

- Example:
  - Alice’s level is secret, Bob’s level is unclassified, Carol’s level is classified
  - Memo1 is classified, and memo2 is top secret
- The *-property specifies that:
  - Bob and Carol can write to memo1, since its level (classified) is not lower than theirs
  - Alice’s level is secret, so she is not permitted to write to memo1
  - Alice, Bob, and Carol are all at a lower level than memo2 and can therefore write to it

The Bell-LaPadula Model (cont)

- This information flow policy controls both direct and indirect paths through which information can flow
- Example: Bob cannot learn the contents of memo1
  - Bob (unclassified) is not allowed to read memo1 (classified) himself
  - Alice (secret) is allowed to read memo1 (classified)
    - Alice is not allowed to write anything but secret or top secret documents which Bob cannot read
    - Bob cannot read anything that Alice writes so Alice cannot tell Bob the contents of memo1
    - Problem: Alice may be able to communicate with Bob without writing any files

The Bell-LaPadula Model (cont)

- Suppose that Bob can observe some aspect of the system that Alice can control
- Example: Bob can determine whether or not Alice is logged in to the system
- Bob and Alice agree that:
  - Every minute Alice spends logged in will represent a 1
  - Every minute Alice spends logged out will represent a 0
- Alice could read memo1 and then begin transmitting it, one bit every minute, to Bob using their **covert channel**
  - There are many other covert channels available on most time-sharing systems
  - Eliminating all possible covert channels can be challenging

Summary

- Important components of computer security:
  - User authentication – determine the identity of an individual accessing the system
    - Knowledge-based, token-based, and biometrics
  - Access control policies – stipulate what actions a given user is allowed to perform on the system
    - Discretionary: access control matrix
    - Mandatory: the Bell-LaPadula model
  - Access control mechanisms – enforce the system's policy
    - Discretionary: access lists, capabilities

Overview

- Computer security threats
  - Unintentional:
    - Coding faults
    - Operational faults
    - Environmental faults
  - Intentional (malicious code):
    - Trojan horses
    - Trap doors
    - Viruses
    - Worms

Physical Security

- **Physical security** entails restricting access by physical means
  - Examples: locked doors and human guards
- Neglecting physical security can undermine other security mechanisms
  - Example: a system with an superb file-protection mechanism
    - The disk drive that stores the filesystem is publicly accessible
    - An attacker could attach his own computer to the drive and read its contents

Human Factors

- **Human factors**
- Users can undermine system security through their naïvete, laziness, or dishonesty
  - Users of a system should also be educated about its security mechanisms so that they are unlikely to accidentally undermine them
  - Users of a system should be screened so that they are unlikely to purposely abuse the system privileges they are given
    - People who have exhibited a pattern of dishonest behavior in the past are risky users

Program Security

- **Program security** requires that the programs that run on a computer system must be:
  - Written correctly (coding faults)
  - Installed and configured properly (operational faults)
  - Used in the manner in which they were intended (environmental faults)
  - Properly behaved (malicious code)
- Flaws in any of these areas may be discovered and exploited by attackers

Program Security (cont)

- **Coding faults** – program bugs that can be exploited to compromise system security
- Examples:
  - **Condition validation errors** – a requirement is either incorrectly specified or incompletely checked
  - **Synchronization errors** – operations are performed in an improper order

Condition Validation Error - Example

- The *uux* (Unix-to-Unix command execution) utility
- Used to execute a sequence of commands on a specified (remote) system
- For security reasons, the commands executed by *uux* should be limited to a set of "safe" commands
  - The *date* command (displays the current date and time) is a safe command and should be allowed
  - The *rm* command (removes files) is not a safe command and should not be allowed

Condition Validation Error – Example (cont)

- Processing *uux* requests:
  - For each command:
    - Check the command to make sure it is in the set of “safe” commands
    - Skip the command’s arguments until a delimiter is reached
  - Example:
    - *cmd1 arg1 arg2 ; cmd2 ; cmd3 arg1*
  - The problem: some implementations did not include the ampersand (&) in the list of delimiters though it is a valid delimiter
  - The result: unsafe commands (e.g. cmd4) could be executed if they followed an ampersand:
    - *cmd2 & cmd4 arg1*

Synchronization Error - Example

- The *mkdir* utility – creates a new subdirectory
  – Creates a new, empty subdirectory (owned by *root*)
  – Changes ownership of the subdirectory from *root* to the user executing *mkdir*
- The problem:
  – If the system is busy, it may be possible to execute a few other commands between the two steps of *mkdir*
  – Example:
    - Delete the new directory after step one and replace it with a link to another file on the system
    - When step two executes it will give the user ownership of the file

Program Security (cont)

- **Malicious code** - programs specifically designed to undermine system security
  - Trojan horses
    - Login spoof
    - Root kits
  - Trap doors
  - Viruses
    - Virus scanning
    - Macro viruses
  - Worms
    - The Morris worm

Trojan Horses

- History – a hollow wooden horse used by the Greeks during the Trojan War
- Today - a **Trojan horse** is a program that has two purposes: one obvious and benign, the other hidden and malicious
- Examples:
  - Login spoof
  - Mailers, editors, file transfer utilities, etc.
  - Compilers

Trojan Horses (cont)

- Examples (cont):
  - Salami
    - Programmer writes bank software that credits interest to customer accounts each month
    - The result of the interest computation on many accounts may not be a whole number of cents
    - Example:
      - 0.25 percent of $817.40 is $2.0435
      - Should be rounded down to $2.04 in interest
    - Programmer instructs program to deposit fractional cents into the programmer's account

Trojan Horses (cont)

- Examples (cont):
  - Root kits
    - A *root kit* is collections of Trojan Horse programs that replace widely-used system utility programs:
      - *ls* and *find* (hides files)
      - *ps* and *top* (hides processes)
      - *netstat* (hides network connections)
    - Goal: conceal the intruder's presence and activities from users and the system administrator

Trap Doors

- **Trap doors** are flaws that designers place in programs so that specific security checks are not performed under certain circumstances
- Example: a programmer developing a computer-controlled door to a bank's vault
  - After the programmer is done the bank will reset all of the access codes to the vault
  - However, the programmer may have left a special access code in his program that always opens the vault

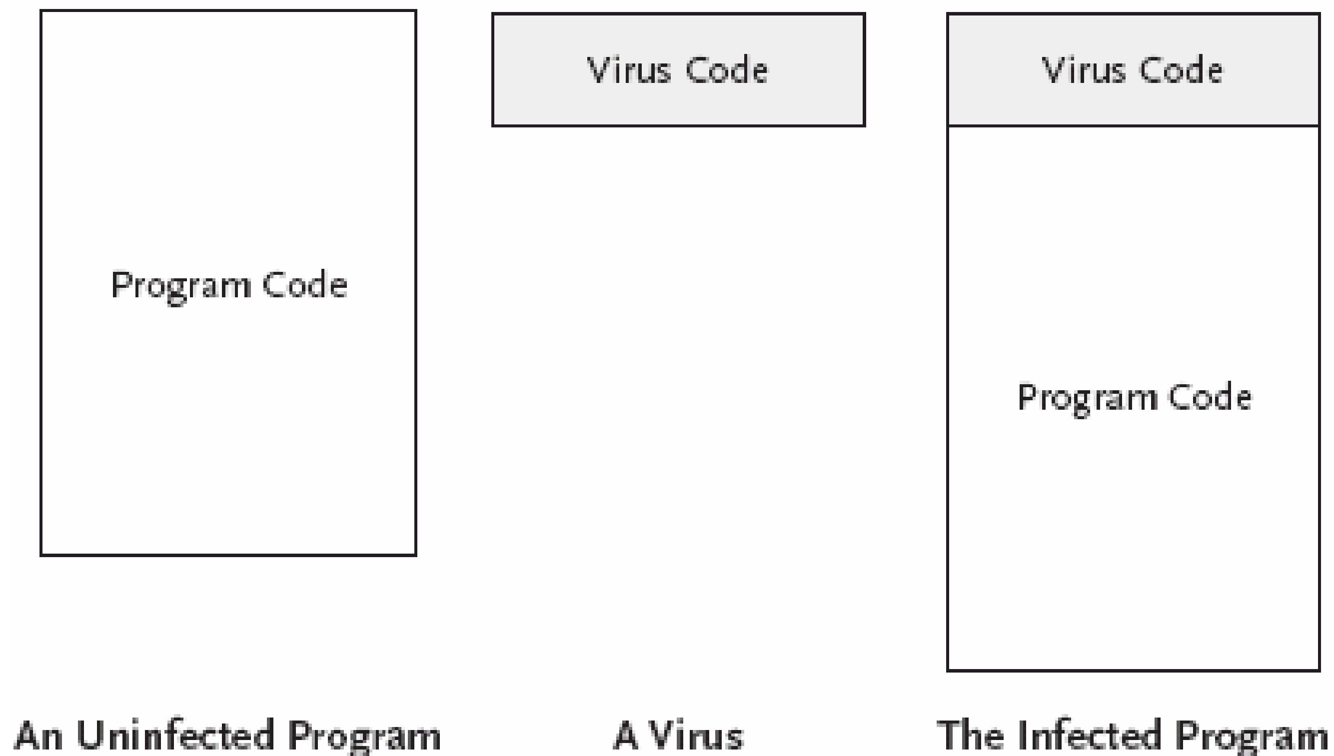
Viruses

- A **virus** is a fragment of code created to spread copies of itself to other programs
- Require a **host** (typically a program):
  - In which to live
  - From which to spread to other hosts
- A host that contains a virus is said to be **infected**
  - A virus typically infects a program by attaching a copy of itself to the program
- Goal: spread and infect as many hosts as possible

Viruses (cont)

- Virus may prepend its instructions to the program's instructions
  - Every time the program runs, virus' code executes
    - **Infection propagation** – mechanism to spread infection to other hosts
    - **Manipulation routine** – (optional) mechanism to perform other actions:
      - Displaying a humorous message
      - Subtly altering stored data
      - Deleting files
      - Killing other running programs
      - Causing system crashes
      - Etc.

Viruses (cont)



Defending Against Computer Viruses

- **Virus scanning** programs check files for signatures of known viruses
  - **Signature =** some unique fragment of code from the virus that appears in every infected file
- Problems:
  - **Polymorphic viruses** that change their appearance each time they infect a new file
    - No easily recognizable pattern common to all instances of the virus
  - New viruses (and modified old viruses) appear regularly
    - Database of viral signatures must be updated frequently

Macro Viruses

- More than just programs can serve as hosts for viruses
- Examples: spreadsheet and word processor programs
  - Usually include a macro feature that allows a user to specify a series of commands to be executed
  - Macros provide enough functionality for a hacker to write a **macro virus**:
    - Executed every time an infected document is opened
    - Has an infection propagation mechanism
    - May have a manipulation routine
  - Example: Microsoft Word:
    - *AutoOpen* macro – run automatically whenever the document containing it is opened
    - *AutoClose* macro – run automatically whenever the document containing it is closed

The Melissa Macro Virus

- March, 1999, Exploited Microsoft Word macros
- Spread by e-mail:
  - Victim received an e-mail message with the subject line "Important Message From NAME"
  - Infected Word document as an attachment:
  - When an infected document was opened:
    - Virus attempted to infect other documents
    - E-mail a copy of an infected document to up to fifty other people
      - E-mail addresses of the new victims were taken from a user's Outlook address book
      - Value to use for NAME in the subject line was read from the Outlook settings

Melissa's Impact

- In three days, infected more than 100,000 computers
- Some sites received tens of thousands of e-mail messages in less than an hour
  - System performance degradation or Mail server crash
- Besides spreading the virus:
  - Modified settings in Microsoft Word to conceal its presence
  - Occasionally modified \ contents of the documents that it infected
  - Occasionally sent sensitive documents without owner's knowledge

Worms

- Virus = a program fragment
- **Worm =** a stand-alone program that can replicate itself and spread
- Worms can also contain manipulation routines to perform other actions:
  - Modifying or deleting files
  - Using system resources
  - Collecting information
  - Etc.

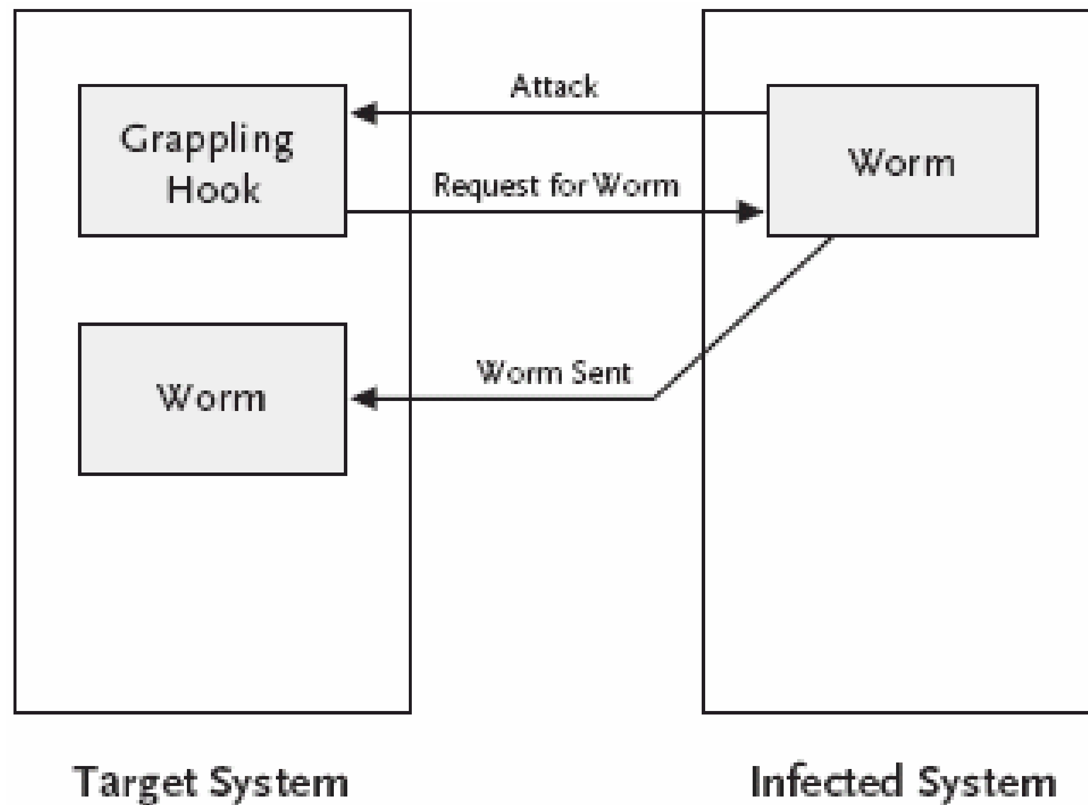
The Morris Worm

- November, 1988
- Created by Robert Morris
- Brought down thousands of the ~60,000 computers then attached to the Internet
  - Academic, governmental, and corporate
  - Suns or VAXes running BSD UNIX

Operation of the Morris Worm

- Used four different attack strategies to try to run a piece of code called the **grappling hook** on a target system
- When run, the grappling hook:
  - Made a network connection back to the infected system from which it had originated
  - Transferred a copy of the worm code from the infected system to the target system
  - Started the worm running on the newly infected system

The Morris Worm's Grappling Hook



Attack Strategy #1: Exploiting *sendmail*

- Many versions of *sendmail* had a debug option
  - Allowed an e-mail message to specify a program as its recipient
- Named program ran with the body of the message as input
- The worm created an e-mail message:
  - Contained the grappling hook code
  - Invoked a command to strip off the mail headers
  - Passed the result to a command interpreter

Attack Strategy #2: Exploiting the *finger* daemon

- Finger daemon, *fingerd*, is a remote user information server
  - Which users currently logged onto the system
  - How long each has been logged on
  - The terminal from which they are logged on
  - Etc.
- A **buffer overflow** bug in *fingerd* on VAXes allowed the worm to execute the grappling hook code

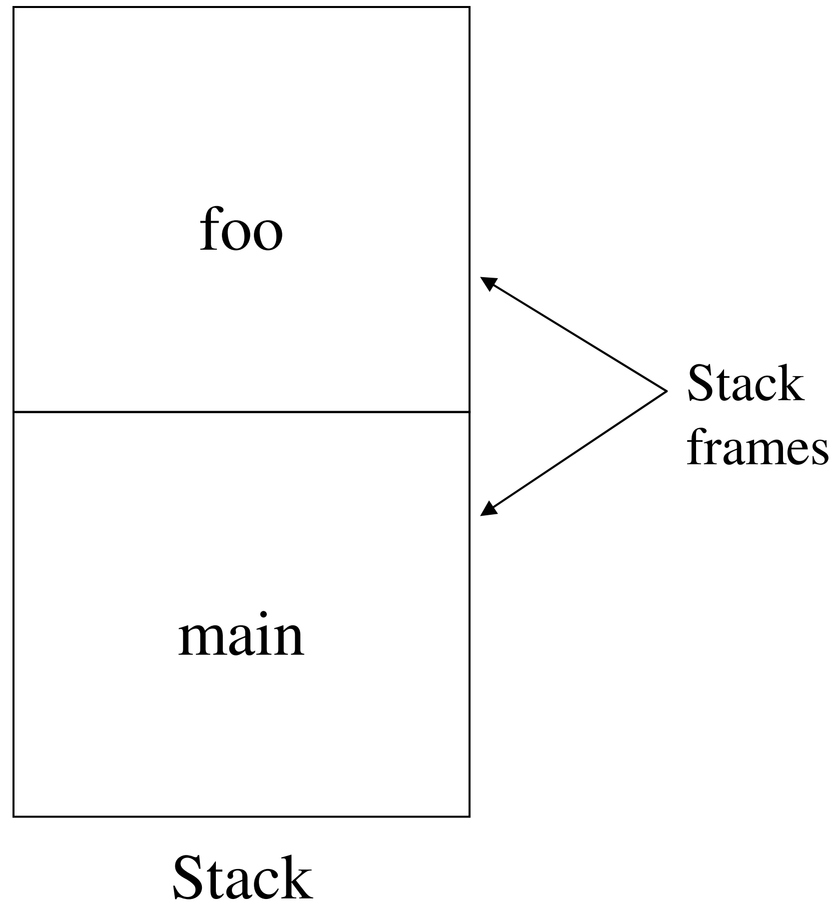
Buffer Overflows

- A program's *stack segment*:
  - Temporary working space for the program
  - Example: Subroutines

```
int foo(int P1, int P2) /* subroutine “foo” */
{
    int L1, L2; /* local variables L1 and L2 */
    L1 = P1 + P2;
    return(L1); /* return value */
}

int main() /* main program */
{
    …
    x = foo(1,2); /* call to subroutine “foo” */
    …
}
```

Stack Frames



Stack Frames (cont)

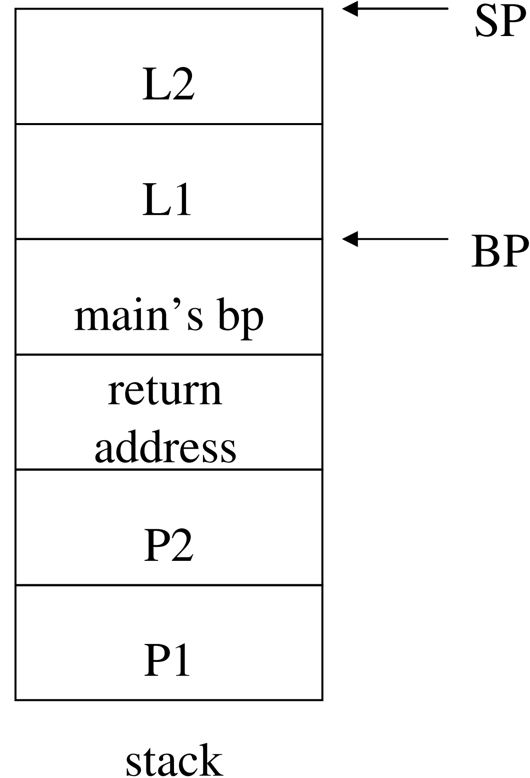
- A stack frame contains the corresponding routine's:
  - Parameters
  - Return address (i.e. next instruction to execute upon completion)
  - Saved registers
  - Local variables
- Many architectures have registers:
  - SP, the *stack pointer*, points to the top of the stack
  - BP, the *base pointer*, points to a fixed location within the frame
    - Used to reference the procedure's parameters and local variables

Stack Frames (cont)

- The *main* routine calls *foo*:
  - *foo*'s parameters are first pushed onto the stack
  - The next instruction in *main* to execute after *foo* finishes, the return address, is pushed
  - Control is transferred to *foo*
  - *foo*'s prologue:
    - Save caller's (*main*'s) base pointer
    - Set callee's (*foo*'s) *bp* equal to the current *sp*
    - Increment *sp* to reserve space on the stack for foo's local variables

Stack Frames (cont)

- *foo*'s stack frame at the after the completion of the prologue:

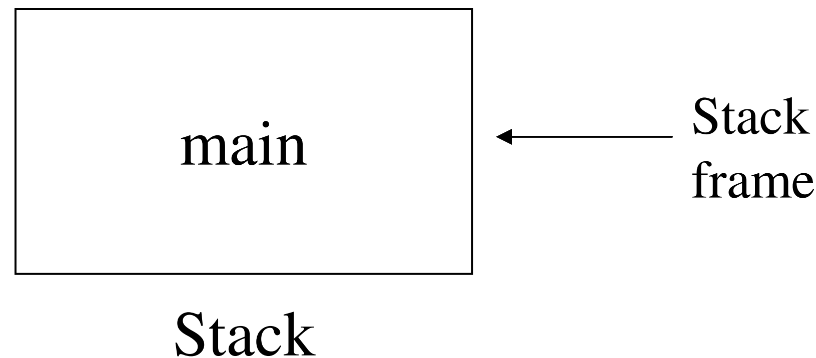


Stack Frames (cont)

- The execution of *foo*:
  - $P1 = BP-4$
  - $P2 = BP-3$
  - $L1 = BP$
  - $L2 = BP+1$
- The statement “L1 = P1 + P2” would performed by the following assembly language instruction:
  - add BP-4, BP-3, BP // adds first two arguments and stores the result in the third

Stack Frames (cont)

- *foo*'s epilogue cleans up the stack and returns control to the caller:
  - Caller's (*main*'s) *bp* is placed back into the *bp* register
  - The return address is placed into the *ip* (instruction pointer) register
  - The stack pointer is decremented to remove the callee's frame from the stack



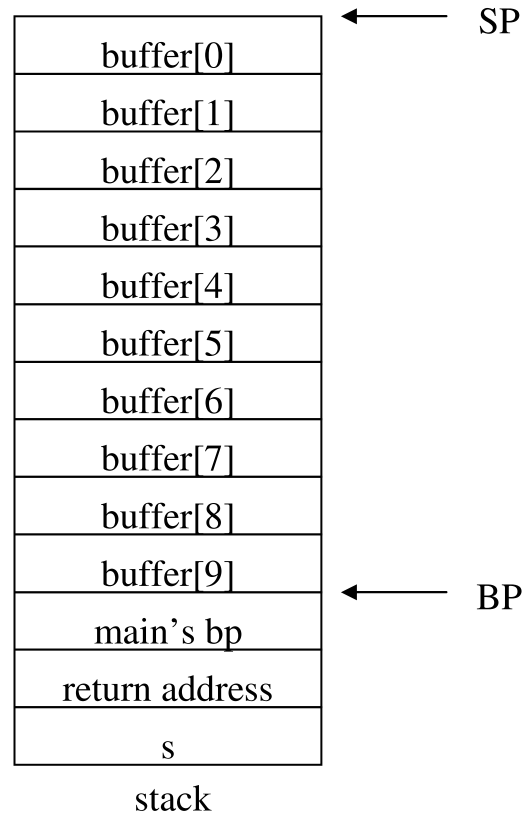
A Buffer Overflow

```
int foo(char *s) /* subroutine “foo” */
{
   char buffer[10]; /* local variable*/
   strcpy(buffer,s);
}

int main() /* main program */
{
   char name[]=”ABCDEFGHIJKL”;
   foo(name); /* call to subroutine “foo” */
}
```

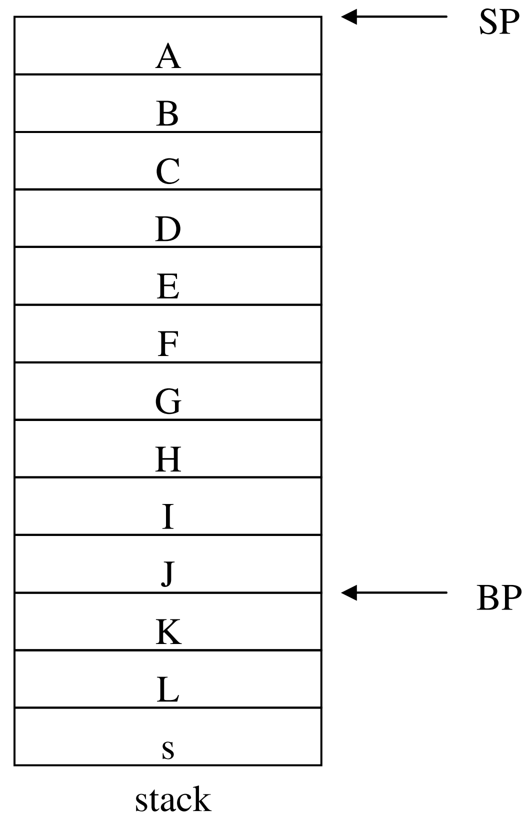
A Buffer Overflow (cont)

- *foo*'s stack frame after prologue:



A Buffer Overflow (cont)

- Stack after execution of *foo* (but before the epilogue):



A Buffer Overflow (cont)

- The string overflowed *foo*'s buffer:
  - Overwrote *main*'s bp
  - Overwrote the return address with 'L' = 89 (ASCII)
- When *foo* finishes control will be transferred to the instruction at address 89
  - Error
- The Morris worm sent a specially crafted 243-byte string to the finger daemon:
  - Overflowed a buffer and overwrote the return address
  - The *fingerd* executed the */bin/sh* program which executed the grappling hook code

Attack Strategy #3: Exploiting *rsh*

- *rsh* = “remote shell”
  - Allows users to execute commands on a remote host from a machine that the remote host trusts
    - */etc/hosts.equiv*
    - *.rhosts*
- The worm used *rsh* to run the grappling hook code on remote computers that trusted an infected machine

Attack Strategy #4: Exploiting *rexec*

- *rexec* = remote execution
  - Protocol that enables users to execute commands remotely
    - Must specify:
      - A host and a valid username and password for that host
- The worm attempted to crack passwords on each computer that it infected so that it could use *rexec* to infect other hosts
  - No password
  - The username
  - The username appended to itself
  - The user's last name or nickname
  - The user's last name reversed
  - Dictionary attack using 432-word dictionary carried with the worm
  - Dictionary attack using ~25,000 words in */etc/dict/words*

Operation of the Worm

- Attempted to camouflage its activity:
  - Changed its process name to *sh*
  - Erased its argument list after processing it
  - Deleted its executable from the filesystem once it was running
  - Various steps to make sure that a *core* file would not be generated
  - Spent most time sleeping
  - Forked every three minutes, parent process exited and the child continued
    - Changed the worm's process identification number (pid) often
    - Prevent the worm from accumulating too much CPU time
  - All constant strings inside the worm were XORed character-by-character with the value 81_{16}
  - Used a simple challenge and response mechanism to determine whether or not a machine it had just infected was already running a copy of the worm
    - Immortal – one in seven times this check was not performed

Aftermath

- The worm spread quickly and infected a large percentage of the computers connected to the Internet
- Noticed within hours
- Took days for researchers to discover how the worm worked and how to stop it
- In 1990, Morris was convicted by a federal court of violating the Computer Crime and Abuse Act of 1986:
  - Three years of probation
  - Four hundred hours of community service
  - $10,050 fine

Summary

- **Program security** requires that the programs that run on a computer system be:
  - Written correctly
  - Installed and configured properly
  - Used in the manner in which they were intended
  - Do not behave maliciously
    - Trojan horses - a program that has two purposes: one obvious and benign, the other hidden and malicious
    - Viruses - a fragment of code created to spread copies of itself to other programs
    - Worms - a stand-alone program that can replicate itself and spread