BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Protection and Security

Components of Computer Security

- **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.

The Reference Monitor

- Computer systems are composed of many diverse objects that can be employed by users to accomplish tasks (e.g. CPU, memory segments, files, printers, etc.).
- It is the job of a **reference monitor** to control access to system objects.
- The 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 (e.g. badges at a military installation).
- 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** – users prove their identity through something that they know.
    - Example: passwords.
  - **Token-based** – users prove their identity through something they possess.
    - Example: passport.
  - **Biometric** – users prove their identity through a unique physiological characteristic.
    - Example: fingerprint.

Passwords

- Passwords are widely-used for user authentication.
- **Advantages:**
  - Easy to use, understood by most users.
  - Require no special equipment.
  - Offer an adequate degree of security in many environments.
- **Disadvantages:**
  - Users tend to choose passwords that are easy to guess.
  - Many password-cracking tools are 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

- A ***one-way hash*** of the password, *h*(password), is stored in the table (not the password itself):
  - *h*(dumptruck) = JFNXPEMD
  - *h*(baseball) = WSAWFFVI

Username	Uid	Hash
Alice	12	JFNXPEMD
Bob	7	WSAWFFVI

Using Passwords and One-Way Functions

- 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 (WSAWFFVI tells us Bob's password is baseball).

Combating Dictionary Attacks

- **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.
  - Between 20 and 40 percent of the passwords on a typical system can be cracked in this way.
- **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

- 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 are much harder (though not impossible).

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.

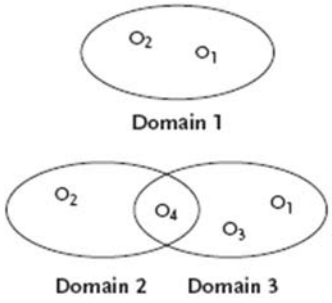
The Access Control Matrix

- Suggested by Butler Lampson.
- 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.**
  - 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.**

Protection Domains - Example

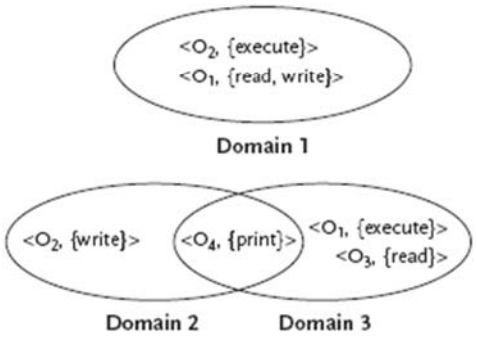
Assume:

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



Protection Domains

- 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

- 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}

Physical Security

- **Physical security** entails restricting access to some object by physical means.
  - Examples: locked doors and human guards.
- Neglecting physical security can undermine other security mechanisms that protect a system.
  - 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** - the users of computer systems impact security.
- 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.
    - Explain to users why certain passwords are weak or help them choose strong ones.
  - 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

- **Malicious code (malware)** - programs specifically designed to undermine the security of a system.
  - Trojan horses
    - Login spoof
    - Root kits
  - Trap doors
  - Viruses
    - Virus scanning
  - Worms
    - The Morris worm, Code Red, Code Red II, Nimda, Slammer

Trojan Horses

- History – a hollow wooden horse used by the Greeks during the Trojan War.
- Today - a **Trojan horse (trojan)** 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.

Root Kits

Definition: A **root kit** is a collection of trojans to replace widely used system utility programs in order to conceal the activities of an intruder.

Example: You break into a system, you upload some files and install services to create a backdoor. The system administrator can find evidence of your intrusion by listing the files on the computer, by listing the running processes, and by looking at system logs. What do you do to cover your tracks?

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.

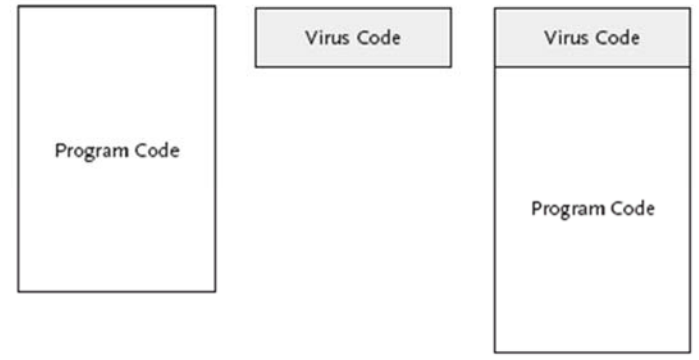
Viruses

- 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 is run the virus' code is executed:
    - **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)



An Uninfected Program A Virus The Infected Program

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.

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.

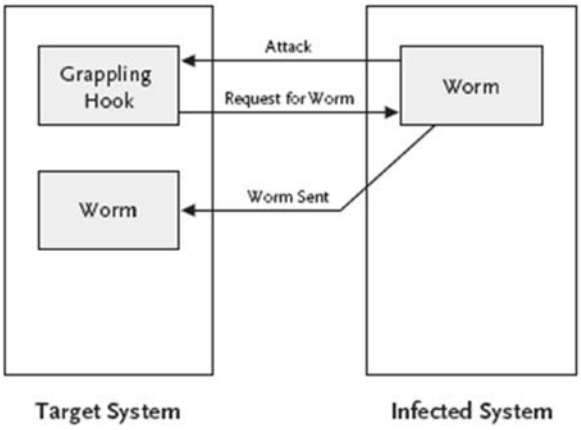
The Morris Worm

- Appeared in November, 1988.
- Created by a Computer Science graduate student.
- 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
Target: ***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
Target: the ***finger*** daemon

- The 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.

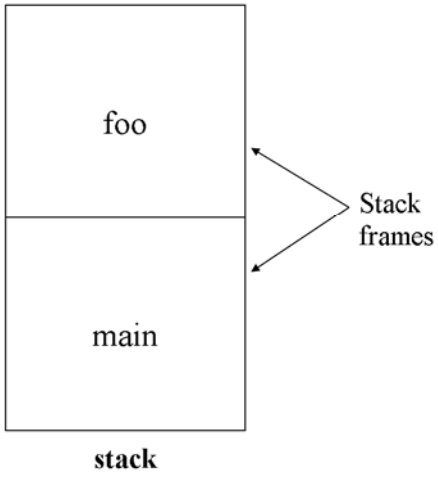
Buffer Overflows

- 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)

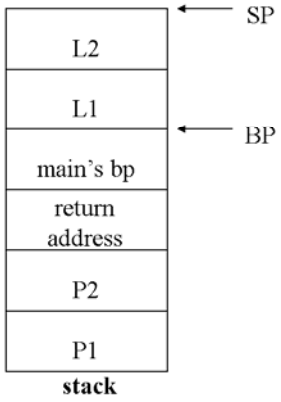
- A stack frame contains the corresponding routine's:
  - Parameters.
  - Return address (i.e. next instruction to execute upon completion).
  - Saved registers.
  - Local variables.
- Many architectures have registers:
  - SP, the *stack pointer*, points to the top of the stack.
  - BP, the *base pointer*,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

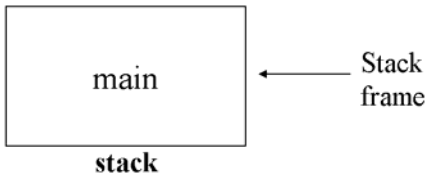


Stack Frames (cont)

- 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.



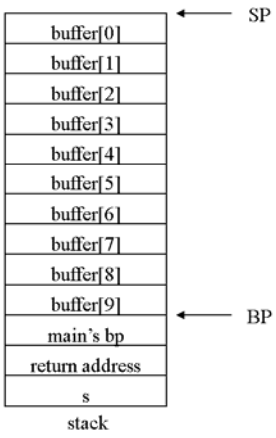
A Buffer Overflow

```
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” */
}
```

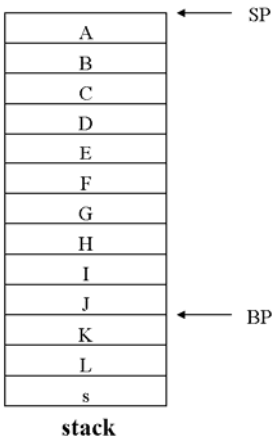
A Buffer Overflow (cont)

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
Target: *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
Target: *rexec*

- *rexec* = **"remote execution"**
  - Protocol that enables users to execute commands remotely
    - Must specify:
      - A host
      - 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

Performed many actions to try 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.

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.