Program security:chp2

Secure programs

- Pgm is secure if we trust that it provides/enforces:
  - Confidentiality
  - Integrity
  - Availability

Types of flaws:

- Some flaws are *intentional*
  - Malicious flaws are intentionally inserted to attack
    - If it's meant to attack some particular system, we call it a targeted malicious flaw
  - Nonmalicious (but intentional) flaws are often features that are meant to be in the system
    - are correctly implemented,
    - but can cause a failure when used by an attacker
- Most security flaws are caused by *unintentional* program errors

Nonmalicious program errors

- nonmalicious program errors/flaws
  - Buffer overflows
  - Incomplete mediation
  - TOCTTOU (time of check to time of use)errors (race conditions)

Key Concepts of Buffer Overflow

- This error occurs when there is more data in a buffer than it can handle, causing data to overflow into adjacent storage.
- This vulnerability can cause a system crash or, worse, create an entry point for a cyberattack.
- C and C++ are more susceptible to buffer overflow.
- Secure development practices should include regular testing to detect and fix buffer overflows. These practices include automatic protection at the language level and bounds-checking at run-time.

- A buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. A buffer overflow, or buffer overrun, occurs when more data is put into a fixed-length buffer than the buffer can handle. The extra information, which has to go somewhere, can overflow into adjacent memory space, corrupting or overwriting the data held in that space. This overflow usually results in a system crash, but it also creates the opportunity for an attacker to run arbitrary code or manipulate the coding errors to prompt malicious actions.
- Many programming languages are prone to buffer overflow attacks. However, the extent of such attacks varies depending on the language used to write the vulnerable program. For instance, code written in Perl and JavaScript is generally not susceptible to buffer overflows. However, a buffer overflow in a program written in C, C++, Fortran or Assembly could allow the attacker to fully compromise the targeted system.

Executing a Buffer Overflow Attack

- Cybercriminals exploit buffer overflow problems to alter the execution path of the application by overwriting parts of its memory.
- The malicious extra data may contain code designed to trigger specific actions — in effect sending new instructions to the attacked application that could result in unauthorized access to the system.
- Hacker techniques that exploit a buffer overflow vulnerability vary per architecture and operating system.

Buffer Overflow Causes

- Coding errors are typically the cause of buffer overflow.
- Common application development mistakes that can lead to buffer overflow include failing to allocate large enough buffers and neglecting to check for overflow problems.
- These mistakes are especially problematic with C/C++, which does not have built-in protection against buffer overflows. Consequently, C/C++ applications are often targets of buffer overflow attacks.

Buffer Overflow Attack Example

- In some cases, an attacker injects malicious code into the memory that has been corrupted by the overflow.
- In other cases, the attacker simply takes advantage of the overflow and its corruption of the adjacent memory.
- For example, consider a program that requests a user password in order to grant the user access to the system. In the code below, the correct password grants the user root privileges. If the password is incorrect, the program will not grant the root privileges.

```
printf ("\n Correct Password \n");
pass = 1;
}
if(pass)
{
/* Now Give root or admin rights to user*/
printf ("\n Root privileges given to the user \n");
}
return 0;
```

- However, there might be a possibility of buffer overflow in a program because the gets() function does not check the array bounds.
- Here is an example of what an attacker could do with this coding error:

```
$ ./bfrovrflw
Enter the password :
hhhhhhhhhhhhhhhhhhhhh
Wrong Password
Root privileges given to the user
```

- In the above example, the program gives the user root privileges, even though the user entered an incorrect password.
- In this case, the attacker supplied an input with a length greater than the buffer can hold, creating buffer overflow, which overwrote the memory of “pass.” Therefore, despite the incorrect password, the value of “pass” became non zero, and the attacker receives root privileges.

- **Buffer Overflow Solutions**
- To prevent buffer overflow, developers of C/C++ applications should avoid standard library functions that are not bounds-checked, such as gets, scanf and strcpy.
- In addition, secure development practices should include regular testing to detect and fix buffer overflows. The most reliable way to avoid or prevent buffer overflows is to use automatic protection at the language level.
- Another fix is bounds-checking enforced at run-time, which prevents buffer overrun by automatically checking that data written to a buffer is within acceptable boundaries.

Incomplete mediation

mediation

  - Web-based applications are a common example
- An application needs to ensure that what user has entered constitutes a meaningful request
  - This is called mediation

Incomplete mediation

- Incomplete mediation occurs when the application accepts incorrect data from user

- In C program a function strcpy(buffer, input) copies the contents of the input string input to the array buffer. A buffer overflow will occur if the length of input is greater than the length of buffer.
- To prevent such a buffer overflow, the program validates the input by checking the length of input before attempting to write it to buffer. Failure to do so is an example of incomplete mediation.
- Example: Input data to a web form is often transferred to the server by embedding it in a URL.

Suppose the input is validated on the client before constructing the required URL say

www.kt280.com/orders/final&custID=111&num=55A&qty=5&price=60&shipping=4&total=300

- This URL is interpreted to mean that the customer with ID number 111 has ordered 5 books of item number 55, at a cost of rs60 each, with a rs5 shipping charge, giving a total cost of rs300.
- Since the input is checked on the client, the developer of the server software believes it would be wasted effort to check it again on the server.
- However, instead of using the client software, Trudy can directly send a URL to the server. Suppose Trudy sends the following URL to the server:

www.kt280.com/orders/final&custID=111&num=55A&qty=5&price=60&shipping=4&total=30

- If the server doesn't bother to validate the input, Trudy can obtain the same order as above, but for the bargain basement price of rs30 instead of the legitimate price of rs300.
- There have been numerous buffer overflows in the Linux kernel, and most of these were due to incomplete mediation.
- There are tools available to help find likely cases of incomplete mediation, but they are not a cure-all since this problem can be subtle, and therefore difficult to detect automatically.

Defences against incomplete mediation

- For values entered by the user
  - Always do very careful checks on the values of all fields
  - These values can potentially contain completely arbitrary 8-bit data and be of any length
- For state stored by the client:
  - Make sure the client has not modified the data in any way

Time-Of-Check To Time-Of-Use errors

- TOCTTOU (“TOCK-too”) errors
  - Also known as “race condition” errors

In software development, **time of check to time of use** (**TOCTTOU** or **TOCTOU**, pronounced "*tock too*") is a class of software bugs caused by changes in a system between the *checking* of a condition (such as a security credential) and the *use* of the results of that check. This is one example of a race condition.

- This weakness can be security-relevant when an attacker can influence the state of the resource between check and use. This can happen with shared resources such as files, memory, or even variables in multithreaded programs.

•Malicious Code

- Malicious code or rogue pgm is written to exploit flaws in pgms

Malicious code can change

- data
- other programs

- Outline for this Subsection:

• General-Purpose Malicious Code (incl. Viruses)

• Targeted Malicious Code

virus

- A virus is a computer code or program, which is capable of affecting your computer data badly by corrupting or destroying them.
- Computer virus has the tendency to make its duplicate copies at a swift pace, and also spread it across every folder and damage the data of your computer system.
- A computer virus is actually a malicious software program or "malware" that, when infecting your system, replicates itself by modifying other computer programs and inserting its own code.

- Following are the other malicious code……
- Worms
- This is a computer program that replicates itself .Unlike a computer virus, it is self-contained and hence does not need to be part of another program to propagate itself.
- Trojan Horse
- A Trojan Horse is also a sort of destructive program that remains disguised in a normal software program. It is not exactly a virus, as it cannot replicate itself. However, there is possibility that virus program may remain concealed in the Trojan Horse.
- Bombs
- It is similar to Trojan Horse, but Logic bombs have some speciality; these include a timing device and hence it will go off only at a particular date and time.

How Does Virus Affect?

- Let us discuss in what ways a virus can affect your computer system. The ways are mentioned below –
- By downloading files from the Internet.
- During the removable of media or drives.
- Through pen drive.
- Through e-mail attachments.
- Through unpatched software & services.
- Through unprotected or poor administrator passwords.

Impact of Virus

- Let us now see the impact of virus on your computer system −
- Disrupts the normal functionality of respective computer system.
- Disrupts system network use.
- Modifies configuration setting of the system.
- Destructs data.
- Disrupts computer network resources.
- Destructs of confidential data.

Virus Detection

- The most fundamental method of detection of virus is to check the functionality of your computer system; a virus affected computer does not take command properly.
- However, if there is antivirus software in your computer system, then it can easily check programs and files on a system for virus signatures.

Virus Preventive Measures

- A computer system can be protected from virus through the following –
- Installation of an effective antivirus software.
- Patching up the operating system.
- Patching up the client software.
- Putting highly secured Passwords.
- Use of Firewalls.

Most Effective Antivirus

- Following are the most popular and effective antivirus from which you can choose one for your personal computer −
- McAfee Antivirus Plus
- Symantec Norton Antivirus
- Avast Pro Antivirus
- Bitdefender Antivirus Plus
- Kaspersky Anti-Virus
- Avira Antivirus
- Webroot Secure Anywhere Antivirus
- Emsisoft Anti-Malware
- Quick Heal Antivirus
- ESET NOD32 Antivirus

Malicious software

- Malicious software (malware) is any software that gives partial to full control of the system to the attacker/malware creator.

- **Virus** – A virus is a program that creates copies of itself and inserts these copies into other computer programs, data files, or into the boot sector of the hard-disk. Upon successful replication, viruses cause harmful activity on infected hosts such as stealing hard-disk space or CPU time.
- **Worm** – A worm is a type of malware which leaves a copy of itself in the memory of each computer in its path.
- **Trojan** – Trojan is a non-self-replicating type of malware that contains malicious code, which upon execution results in loss or theft of data or possible system harm.
- **Adware** – Adware, also known as freeware or pitchware, is a free computer software that contains commercial advertisements of games, desktop toolbars, and utilities. It is a web-based application and it collects web browser data to target advertisements, especially pop-ups.
- **Spyware** – Spyware is infiltration software that anonymously monitors users which enables a hacker to obtain sensitive information from the user's computer. Spyware exploits users and application vulnerabilities that is quite often attached to free online software downloads or to links that are clicked by users.
- **Rootkit** – A rootkit is a software used by a hacker to gain admin level access to a computer/network which is installed through a stolen password or by exploiting a system vulnerability

Targeted Malicious Code

- Targeted = written to attack a particular system, a particular application, and for a particular purpose

- ***Targeted Malicious Code*** – written for a particular system or application with a particular purpose
- Similar to viruses but with the addition of new techniques

- Program Threats
  - Trapdoors
  - Salami Attack
  - Privilege Escalation
  - Man-in-the-Middle
  - Covert Channels

Trapdoors

- A **trapdoor** is an undocumented entry point to a module. The trapdoor is inserted during code development, perhaps to test the module, to provide "hooks" by which to connect future modifications or enhancements or to allow access if the module should fail in the future.
- In addition to these legitimate uses, trapdoors can allow a programmer access to a program once it is placed in production.

- Testing:
- each small component of the system is tested first, separate from the other components, in a step called **unit testing**, to ensure that the component works correctly by itself. Then, components are tested together during **integration testing**, to see how they function as they send messages and data from one to the other.

Source of trapdoors

- ***Stubs and Drivers*** – routines that inject information during testing
- ***Control Stubs*** – used to invoke debugging code

- Stubs and Drivers are two types of test harness. Test harness are the collection of software and test data which is configured so that we can test a program unit by simulating different set of conditions, while monitoring the behavior and outputs.
- Stubs and drivers both are dummy modules and are only created for test purposes.

- Stubs are used in *top down testing approach*, when you have the major module ready to test, but the sub modules are still not ready yet. So in a simple language stubs are "called" programs, which are called in to test the major module's functionality.

- For eg. suppose you have three different modules : **Login, Home, User**. Suppose login module is ready for test, but the two minor modules Home and User, which are called by Login module are not ready yet for testing.
  At this time, we write a piece of dummy code, which simulates the called methods of Home and User. These dummy pieces of code are the stubs.

  On the other hand, Drivers are the ones, which are the "calling" programs. Drivers are used in *bottom up testing approach*. Drivers are dummy code, which is used when the sub modules are ready but the main module is still not ready.

  Taking the same example as above. Suppose this time, the **User and Home** modules are ready, but the Login module is not ready to test. Now since Home and User return values from Login module, so we write a dummy piece of code, which simulates the Login module. This dummy code is then called Driver.

Source of trapdoor (cont)

- Poor **error checking** is another source of trapdoors. A good developer will design a system so that any data value is checked before it is used; the checking involves making sure the data type is correct as well as ensuring that the value is within acceptable bounds. But in some poorly designed systems, unacceptable input may not be caught and can be passed on for use in unanticipated ways.

Causes of Trapdoors

trapdoors can persist in production programs because the developers:

- *forget* to remove them
- intentionally leave them in the program for *testing*
- intentionally leave them in the program for *maintenance* of the finished program, or
- intentionally leave them in the program as a *covert means of access* to the component after it becomes an accepted part of a production system

- It is important to remember that the fault is not with the trapdoor itself, which can be a very useful technique for program testing, correction, and maintenance. Rather, the fault is with the system development process, which does not ensure that the trapdoor is "closed" when it is no longer needed. That is, the trapdoor becomes a vulnerability if no one notices it or acts to prevent or control its use in vulnerable situations.
- In general, trapdoors are a vulnerability when they expose the system to modification during execution. They can be exploited by the original developers or used by anyone who discovers the trapdoor by accident or through exhaustive trials. A system is not secure when someone believes that no one else would find the hole.

In short

a. Trapdoors (1)

- Original def:

  Trapdoor / backdoor - A hidden computer flaw known to an intruder, or a hidden computer mechanism (usually software) installed by an intruder, who can activate the trap door to gain access to the computer without being blocked by security services or mechanisms.

- A broader definition:

  Trapdoor – an undocumented entry point to a module

  - Inserted during code development
    - For testing
    - As a hook for future extensions
    - As emergency access in case of s/w failure

Trapdoors (2)

- Testing:
  - With stubs and drivers for unit testing
  - Testing with debugging code inserted into tested modules
    - May allow programmer to modify internal module variables
- Major sources of trapdoors:
  - Left-over (purposely or not) stubs, drivers, debugging code
  - Poor error checking
    - E.g., allowing for unacceptable input that causes buffer overflow
  - Undefined opcodes in h/w processors
    - Some were used for testing, some random
- Not all trapdoors are bad
  - Some left purposely w/ good intentions

— facilitate system maintenance/audit/testing

Salami attack

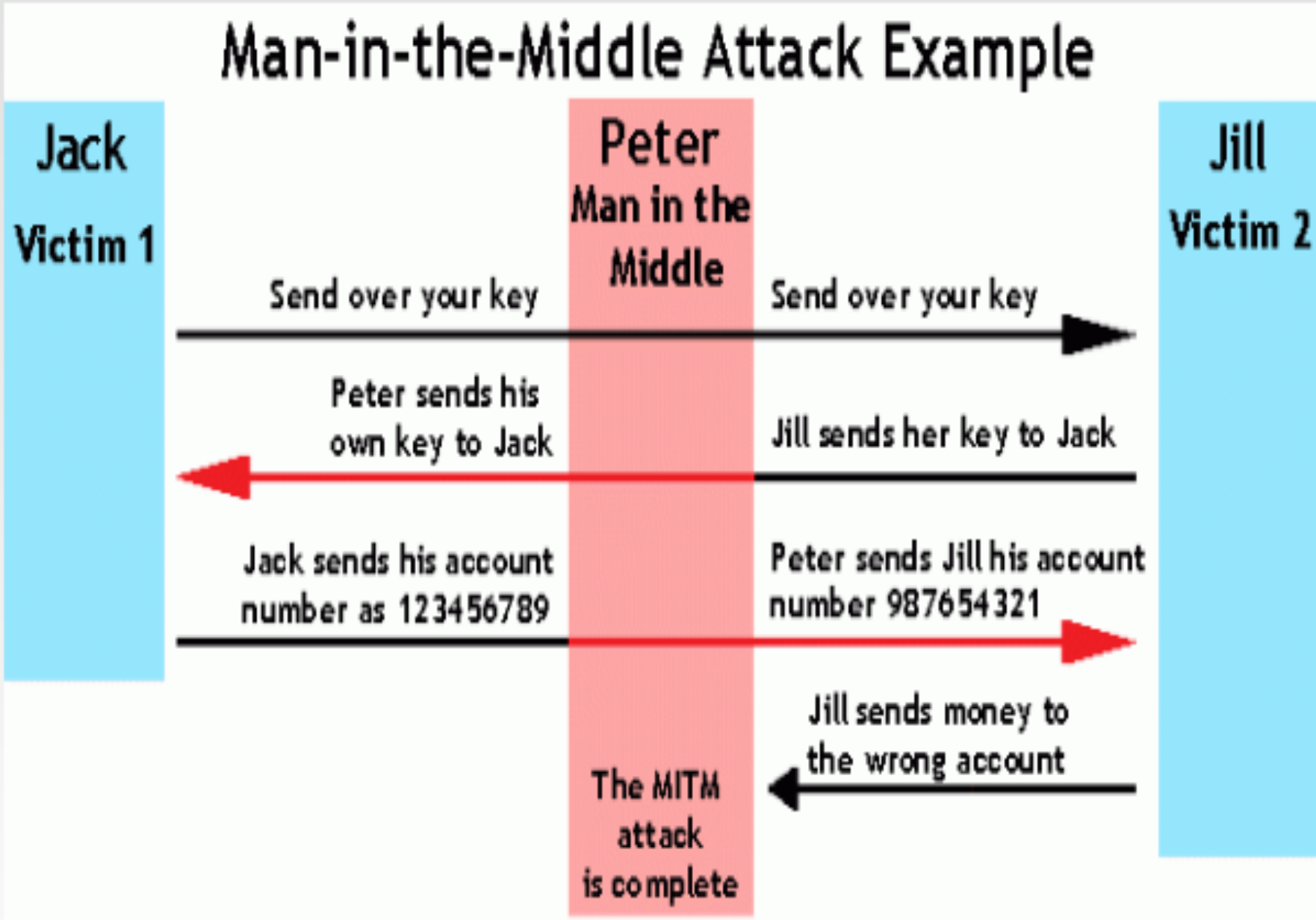
- **What is a Salami Attack?**
- A salami attack is when small attacks add up to one major attack that can go undetected due to the nature of this type of cyber crime. It also known as salami slicing.
- Although salami slicing is often used to carry out illegal activities, it is only a strategy for gaining an advantage over time by accumulating it in small increments, so it can be used in perfectly legal ways as well .
- The attacker uses an online database to seize the information of customers that is bank/credit card details deducting very little amounts from every account over a period of time. The customers remain unaware of the slicing and hence no complaint is launched thus keeping the hacker away from detection.
- Eg: interest computation

privilege escalation

- A privilege escalation attack is a type of network intrusion that takes advantage of programming errors or design flaws to grant the attacker elevated access to the network and its associated data and applications.

- Not every system hack will initially provide an unauthorized user with full access to the targeted system. In those circumstances privilege escalation is required. There are two kinds of privilege escalation: vertical and horizontal.
- Vertical privilege escalation requires the attacker to grant himself higher privileges. This is typically achieved by performing kernel-level operations that allow the attacker to run unauthorized code.
- Horizontal privilege escalation requires the attacker to use the same level of privileges he already has been granted, but assume the identity of another user with similar privileges. For example, someone gaining access to another person's online banking account would constitute horizontal privilege escalation.

Man in the middle


Man-in-the-Middle Attack Example
Jack
Victim 1
Peter
Man in the Middle
Jill
Victim 2
Send over your key
Send over your key
Peter sends his own key to Jack
Jill sends her key to Jack
Jack sends his account number as 123456789
Peter sends Jill his account number 987654321
Jill sends money to the wrong account
The MITM attack is complete

- Other Forms of Session Hijacking
- Man-in-the-middle is a form of session hijacking. Other forms of session hijacking similar to man-in-the-middle are:
- Sidejacking - This attack involves sniffing data packets to steal session cookies and hijack a user's session. These cookies can contain unencrypted login information, even if the site was secure.
- Evil Twin - This is a rogue Wi-Fi network that appears to be a legitimate network. When users unknowingly join the rogue network, the attacker can launch a man-in-the-middle attack, intercepting all data between you and the network.
- Sniffing - This involves a malicious actor using readily available software to intercept data being sent from, or to, your device.

Covert channels

- any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy.

- **covert channel** is a type of attack that creates a capability to transfer information objects between processes that are not supposed to be allowed to communicate by the computer security policy.

- Traffic of a covert channel is
  - Visible
  - Within known protocols
  - Looks like normal traffic

- A covert channel is created by a sender process that modulates some condition (such as free space, availability of some service, wait time to execute) that can be detected by a receiving process.
- two kinds of covert channels:
- Storage channels - Communicate by modifying a "storage location", such as a hard drive.
- Timing channels - Perform operations that affect the "real response time observed" by the receiver.

example

- Data hiding in TCP/IP Protocol suite by covert channels
- Data hiding in LAN environment by covert channels

CONTROLS AGAINST PROGRAM THREATS

- Prevention from threats
- In this section we look at three types of controls: developmental, operating system, and administrative.

Developmental control

- Many controls can be applied during software development to figure out and fix problems. So let us begin by looking at the nature of development itself, to see what tasks are involved in specifying, designing, building, and testing software.

- **The Nature of Software Development**
- Software development is often considered a solitary effort; a programmer sits with a specification or design and grinds out line after line of code. But in fact, software development is a collaborative effort, involving people with different skill sets who combine their expertise to produce a working product.

Development requires people who can

- *specify* the system, by capturing the requirements and building a model of how the system should work from the users' point of view
- *design* the system, by proposing a solution to the problem described by the requirements and building a model of the solution
- *implement* the system, by using the design as a blueprint for building a working solution
- *test* the system, to ensure that it meets the requirements and implements the solution as called for in the design
- *review* the system at various stages, to make sure that the end products are consistent with the specification and design models
- *document* the system, so that users can be trained and supported
- *manage* the system, to estimate what resources will be needed for development and to track when the system will be done
- *maintain* the system, tracking problems found, changes needed, and changes made, and evaluating their effects on overall quality and functionality

- One person could do all these things. But more often than not, a team of developers works together to perform these tasks. Sometimes a team member does more than one activity; a tester can take part in a requirements review, for example, or an implementer can write documentation. Each team is different, and team dynamics play a large role in the team's success.

Modularity, Encapsulation, and Information Hiding

- a key principle of software engineering is to create a design or code in small, self-contained units, called **components** or **modules**; when a system is written this way, we say that it is **modular**. Modularity offers advantages for program development in general and security in particular.

- If a component is isolated from the effects of other components, then it is easier to trace a problem to the fault that caused it and to limit the damage the fault causes. It is also easier to maintain the system, since changes to an isolated component do not affect other components. And it is easier to see where vulnerabilities may lie if the component is isolated. We call this isolation **encapsulation**.

- **Information hiding** is another characteristic of modular software. When information is hidden, each component hides its precise implementation or some other design decision from the others. Thus, when a change is needed, the overall design can remain intact while only the necessary changes are made to particular components.

- There are several advantages to having small, independent components.

- *Maintenance.* If a component implements a single function, it can be replaced easily with a revised one if necessary. The new component may be needed because of a change in requirements, hardware, or environment. Sometimes the replacement is an enhancement, using a smaller, faster, more correct, or otherwise better module. The interfaces between this component and the remainder of the design or code are few and well described, so the effects of the replacement are evident.

- *Understandability.* A system composed of many small components is usually easier to comprehend than one large, unstructured block of code.

- *Reuse.* Components developed for one purpose can often be reused in other systems. Reuse of correct, existing design or code components can significantly reduce the difficulty of implementation and testing.

- *Correctness.* A failure can be quickly traced to its cause if the components perform only one task each.

- *Testing.* A single component with well-defined inputs, output, and function can be tested exhaustively by itself, without concern for its effects on other modules (other than the expected function and output, of course).

Coupling and cohesion

- A modular component usually has high cohesion and low coupling. By **cohesion**, we mean that all the elements of a component have a logical and functional reason for being there; every aspect of the component is tied to the component's single purpose. A highly cohesive component has a high degree of focus on the purpose; a low degree of cohesion means that the component's contents are an unrelated jumble of actions, often put together because of time-dependencies or convenience.
- **Coupling** refers to the degree with which a component depends on other components in the system. Thus, low or loose coupling is better than high or tight coupling, because the loosely coupled components are free from unwitting interference from other components

Peer Reviews

- We turn next to the process of developing software. Certain practices and techniques can assist us in finding real and potential security flaws (as well as other faults) and fixing them before the system is turned over to the users. Of the many practices available for building what they call "solid software," Pfleeger et al. recommend several key techniques: [PFL01a]
- peer reviews
  - hazard analysis
  - testing
- good design
- prediction
- static analysis
- configuration management
- analysis of mistakes
- Here, we look at each practice briefly, and we describe its relevance to security controls. We begin with peer reviews.

Types of Peer Review

- Review- presented formally
- Walk-Through – creator leads and controls the discussion
- Inspection – formal detailed analysis
- Finding a fault and dealing with it:
  - By learning how, when, and why errors occur
  - By taking action to prevent mistakes
  - By scrutinizing products to find the instances and effects of errors that were missed.

Hazard Analysis/Testing

- ***Hazard Analysis*** – set of systematic techniques to expose potentially hazardous system states.
  - Hazards and Operability Studies
  - Failure Modes and effects analysis
  - Fault tree analysis
- Testing
  - Unit Testing
  - Integration Testing
  - Function Testing
  - Performance Testing
  - Acceptance Testing
  - Installation Testing
  - Regression Testing
  - Black-box Testing
  - Clear-box Testing
  - Independent Testing
  - Penetration Testing

Good Design

- Using a philosophy of fault tolerance
- Having a consistent policy for handling failures
- Capturing the design rationale and history
- Using design patterns

- ***Passive fault detection*** – waiting for a system to fail
- ***Active fault detection*** – construct a system that reacts to a failure

Good Design

- Handling Problems
  - Retrying – restoring the system to previous state and try again
  - Correcting – resorting the system to previous state and correcting some system characteristic before trying again
  - Reporting – restoring and reporting but not trying again

Configuration Management

- Who is making the changes
  - Corrective change
  - Adaptive change
  - Perfective change
  - Preventive change
- ***Configuration Management*** – is the process by which we control changes during development and maintenance
  - Configuration identification
  - Configuration control and change management
  - Configuration auditing
  - Status accounting

c. Operating System Controls for Security (1)

- Developmental controls not always used

OR:

- Even if used, not foolproof

=> Need other, complementary controls, incl. OS controls

- Such OS controls can protect against some pgm flaws

Operating System Controls on Use of Programs

- Trusted Software
  - code has been rigorously developed and analyzed
    - Functional correctness
    - Enforcement of integrity
    - Limited privilege
    - Appropriate confidence level

Operating System Controls for Security (3)

- Key characteristics determining if OS code is trusted
  1) Functional correctness
     - OS code should be consistent
  2) Enforcement of integrity
     - OS keeps integrity of its data and other resources even if presented with flawed or unauthorized commands
  3) Limited privileges
     - OS minimizes access to secure data/resources
     - Trusted pgms must have „need to access” and proper access rights to use resources protected by OS
     - Untrusted pgms can’t access resources protected by OS
  4) Appropriate confidence level
     - OS code examined and rated at appropriate trust level

- Similar criteria used to establish if s/w other than OS can be trusted

- To increasing security if untrusted pgms present:
  1) Mutual suspicion
  2) Confinement
  3) Access log

Operating System Controls on Use of Programs

- Mutual Suspicion
  - assume other program is not trustworthy
- Confinement
  - limit resources that program can access
- Access Log
  - list who access computer objects, when, and for how long

Administrative Controls

- Standards of Program Development
  - Standards of design
  - Standards of documentation, language, and coding style
  - Standards of programming
  - Standards of testing
  - Standards of configuration management
  - Security Audits
- Separation of Duties

Administrative Controls for Security (2)

1) Standards and guidelines for program development
   - Capture experience and wisdom from previous projects
   - Facilitate building higher-quality s/w (incl. more secure)
   - They include:
     - Design S&G – design tools, languages, methodologies
     - S&G for documentation, language, and coding style
     - Programming S&G - incl. reviews, audits
     - Testing S&G
     - Configuration mgmt S&G

2) Security audits
   - Check compliance with S&G
   - Scare potential dishonest programmer from including illegitimate code (e.g., a trapdoor)

Administrative Controls for Security (3)

3) Separation of duties

- Break sensitive tasks into ≥ 2 pieces to be performed by different people (learned from banks)
- Example 1: modularity
  - Different developers for cooperating modules
- Example 2: independent testers
  - Rather than developer testing her own code

e. Conclusions (for Controls for Security)

- Developmental / OS / administrative controls help produce/ maintain higher-quality (also more secure) s/w
- „A good developer who truly understands security will incorporate security into all phases of development."
- Summary:

Control	Purpose	Benefit
Develop-mental	Limit mistakes Make malicious code difficult	Produce better software
OperatingSystem	Limit access to system	Promotes safe sharing of info
Adminis-trative	Limit actions of people	Improve usability, reusability and maintainability

Threat analysis

- *Defn of threat:*
- *"Any circumstance or event with the potential to adversely impact an IS through unauthorized access, destruction, disclosure, modification of data, and/or denial of service."*

Types of Threat

- Following are the most common types of computer threats –
- **Physical damage** – It includes fire, water, pollution, etc.
- **Natural events** – It includes climatic, earthquake, volcanic activity, etc.
- **Loss of services** – It includes electrical power, air conditioning, telecommunication, etc.
- **Technical failures** – It includes problems in equipment, software, capacity saturation, etc.
- **Deliberate type** – It includes spying, illegal processing of data, etc.
- Some other threats include error in use, abuse of rights, denial of actions, eavesdropping, theft of media, retrieval of discarded materials, etc.

- Sources of Threat
- The possible sources of a computer threat may be –
- **Internal** – It includes employees, partners, contractors (and vendors).
- **External** – It includes cyber-criminals (professional hackers), spies, non-professional hackers, activists, malware (virus/worm/etc.), etc.

Common Terms

- Following are the common terms frequently used to define computer threat −
- Virus Threats
- A computer virus is a program designed to disrupt the normal functioning of the computer without the permission of the user.
- Spyware Threats
- Spyware is a computer program that monitors user's online activities or installs programs without user's consent for profit or theft of personal information.
- Hackers
- Hackers are programmers who put others on threats for their personal gain by breaking into computer systems with the purpose to steal, change or destroy information.

- Phishing Threats
- It is an illegal activity through which phishers attempt to steal sensitive financial or personal data by means of fraudulent email or instant messages.

How to Secure Your Computer System from Threats?

- Following are the significant tips through which you can protect your system from different types of threat −
- Install, use, and keep updated Anti-Virus in your system.
- Install, use, and keep updated a Firewall Program.
- Always take backups of your important Files and Folders.
- Use Strong and Typical Passwords.
- Take precaution especially when Downloading and Installing Programs.
- Install, use, and keep updated a File Encryption Program.
- Take precaution especially when Reading Email with Attachments.
- Keep your Children aware of Internet threats and safe browsing.

Threat Analysis

- Threats are agents that violate the protection of information assets and site security policy. Threat analysis identifies for a specific architecture, functionality and configuration. Threat analysis may assume a given level of access and skill level that the attacker may possess. Threats may be mapped to vulnerabilities to understand how the system may be exploited. A mitigation plan is composed of countermeasures that are considered to be effective against the identified vulnerabilities that the threats exploit.
- Attackers who are not technologically sophisticated are increasingly performing attacks on systems without really understanding what it is they are exploiting, because the weakness was discovered by someone else. These individuals are not looking to target specific information or a specific company but rather use knowledge of a vulnerability to scan the entire InternetAttackers who are not technologically sophisticated are increasingly performing attacks on systems without really understanding what it is they are exploiting, because the weakness was discovered by someone else. These individuals are not looking to target specific information or a specific company but rather use knowledge of a vulnerability to scan the entire Internet for systems that possess that vulnerability. The table below, which was developed by NIST [4, p. 14], summarizes potential threat sources:

Threat Source	Motivation	Threat Actions
Cracker	Challenge Ego Rebellion	• System profiling • Social engineering • System intrusion, break-ins • Unauthorized system access

Computer criminal	Destruction of information Illegal information disclosure Monetary gain Unauthorized data alteration	• Computer crime (e.g., cyber stalking) • Fraudulent act (e.g., replay, impersonation, interception) • Information bribery • Spoofing • System intrusion • Botnets • Malware: Trojan, virus, worm, spyware • Spam • Phishing

Terrorist	Blackmail Destruction Exploitation Revenge Monetary gain Political gains	• Bomb • Information warfare • System attack (e.g., distributed denial of service) • System penetration • System tampering
Industrial espionage	Competitive advantage Economic espionage Blackmail	• Economic exploitation • Information theft • Intrusion on personal privacy • Social engineering • System penetration • Unauthorized system access (access to classified, proprietary, and/or technology-related information)

Insiders (poorly trained, disgruntled, malicious, negligent, dishonest, or terminated employees)	Curiosity Ego Intelligence Monetary gain Revenge Unintentional errors and omissions (e.g., data entry errors, programming errors) Wanting to help the company (victims of social engineering) Lack of procedures or training	• Assault on an employee • Blackmail • Browsing of proprietary information • Computer abuse • Fraud and theft • Information bribery • Input of falsified, corrupted data • Interception • Malicious code (e.g., virus, logic bomb, Trojan horse) • Sale of personal information • System bugs • System intrusion • System sabotage