UNIT - 2

Programs and Programming

RAKSHITH P

Assistant Professor , CSE

JSS SCIENCE AND TECHNOLOGY UNIVERSITY

Programs and Programming

- Programs are simple things but they can wield mighty power. Think about them for a minute: Programs are just strings of 0s and 1s, representing elementary machine commands such as move one data item, compare two data items, or branch to a different command.
- Most programs are written in higher-level languages such as Java, C, C++, Perl, or Python; programmers often use libraries of code to build complex programs from pieces written by others.
- But most people are not programmers; instead, they use already written applications for word processing, web browsing, graphics design, accounting, and the like without knowing anything about the underlying program code.

- In this chapter describe security problems in programs and programming. As with the light, a problem can reside anywhere between the machine hardware and the user interface.

- Two or more problems may combine in negative ways, some problems can be intermittent or occur only when some other condition is present, and the impact of problems can range from annoying (perhaps not even perceptible) to catastrophic.

Security failures can result from intentional or nonmalicious causes; both can cause harm.

Unintentional (Nonmalicious) Programming Oversights

- Programs and their computer code are the basis of computing. Without a program to guide its activity, a computer is pretty useless.
- Because the early days of computing offered few programs for general use, early computer users had to be programmers too— they wrote the code and then ran it to accomplish some task.
- Today's computer users sometimes write their own code, but more often they buy programs off the shelf; they even buy or share code components and then modify them for their own uses.
- And all users gladly run programs all the time: spreadsheets, music players, word processors, browsers, email handlers, games, simulators, and more.

- A program flaw can be a fault affecting the correctness of the program's result —that is, a fault can lead to a failure. Incorrect operation is an integrity failing.
- A faulty program can also inappropriately modify previously correct data, sometimes by overwriting or deleting the original data. Even though the flaw may not have been inserted maliciously, the outcomes of a flawed program can lead to serious harm.
- On the other hand, even a flaw from a benign cause can be exploited by someone malicious. If an attacker learns of a flaw and can use it to manipulate the program's behavior, a simple and nonmalicious flaw can become part of a malicious attack.

Benign flaws can be—often are—exploited for malicious impact.

What is a Buffer Overflow?

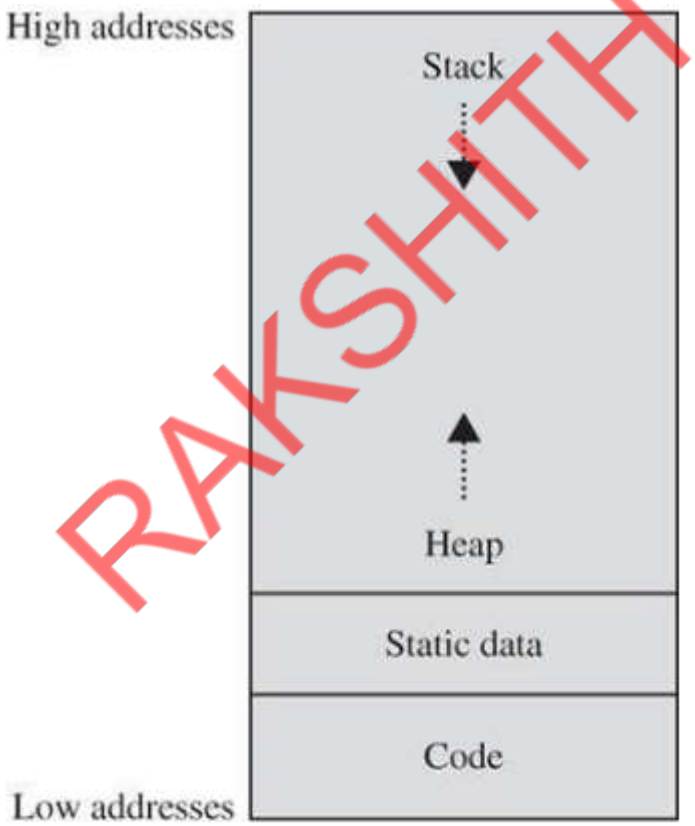
- In a computer program, variables are allocated with fixed-size blocks of memory. After this memory is allocated, the program can store and retrieve data from these locations.
- Buffer overflows occur when the amount of data written to one of these blocks of memory exceeds its size. As a result, memory allocated for other purposes is overwritten, which can have various effects on the program.

Buffer overflows often come from innocent programmer oversights or failures to document and check for excessive data.

Memory Allocation

- Memory is a limited but flexible resource; any memory location can hold any piece of code or data.
- To make managing computer memory efficient, operating systems jam one data element next to another, without regard for data type, size, content, or purpose.
- Users and programmers seldom know, much less have any need to know, precisely which memory location a code or data item occupies.
- Computers use a pointer or register known as a program counter that indicates the next instruction. As long as program flow is sequential, hardware bumps up the value in the program counter to point just after the current instruction as part of performing that instruction.
- Conditional instructions such as IF(), branch instructions such as loops (WHILE, FOR) and unconditional transfers such as GOTO or CALL divert the flow of execution, causing the hardware to put a new destination address into the program counter.
- Changing the program counter causes execution to transfer from the bottom of a loop back to its top for another iteration.
- Hardware simply fetches the byte (or bytes) at the address pointed to by the program counter and executes it as an instruction.

- Show a typical arrangement of the contents of memory, showing code, local data, the heap (storage for dynamically created data), and the stack (storage for subtask call and return data).
- In figure, instructions move from the bottom (low addresses) of memory up; left unchecked, execution would proceed through the local data area and into the heap and stack. Of course, execution typically stays within the area assigned to program code.



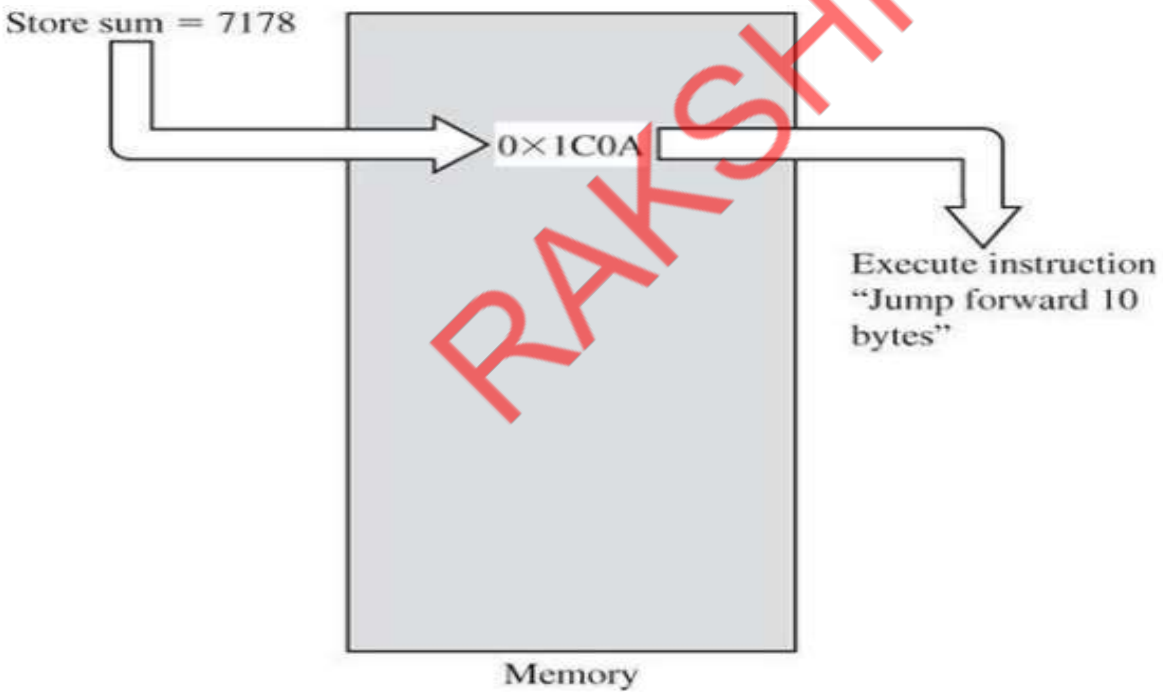
Typical Memory Organization

Code and Data

- Each computer instruction determines how data values are interpreted: An Add instruction implies the data item is interpreted as a number, a Move instruction applies to any string of bits of arbitrary form, and a Jump instruction assumes the target is an instruction.
- But at the machine level, nothing prevents a Jump instruction from transferring into a data field or an Add command operating on an instruction, although the results may be unpleasant. Code and data are bit strings interpreted in a particular way.

In memory, code is indistinguishable from data. The origin of code (respected source or attacker) is also not visible.

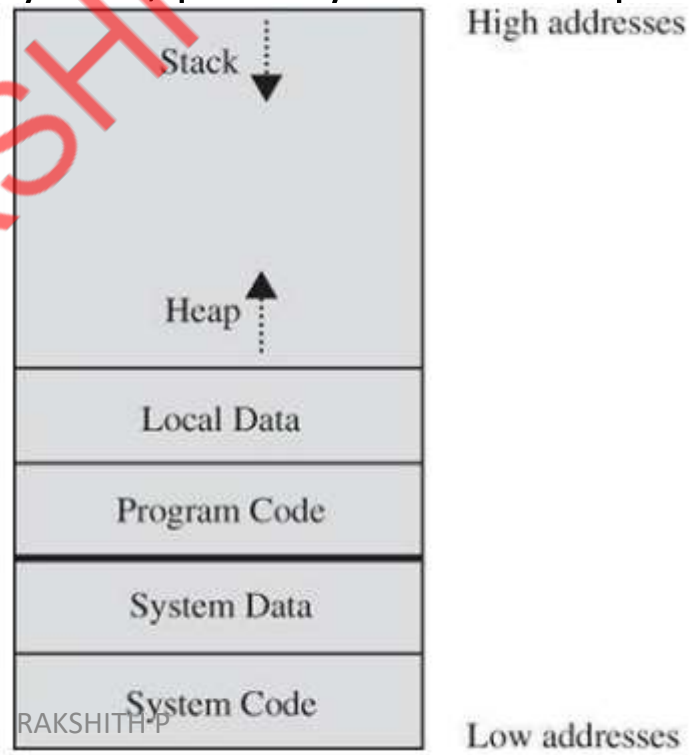
- 0x1C is the operation code for a Jump instruction, and the form of a Jump instruction is 1C displ, meaning execute the instruction at the address displ bytes ahead of this instruction, the string 0x1C0A is interpreted as jump forward 10 bytes.
- But, as shown in Figure , that same bit pattern represents the two-byte decimal integer 7178. So storing the number 7178 in a series of instructions is the same as having programmed a Jump



Bit Patterns Can Represent Data or Instructions

Harm from an Overflow

- The operating system's code and data coexist with a user's code and data. The heavy line between system and user space is only to indicate a logical separation between those two areas; in practice, the distinction is not so solid.
- If the attacker can gain control by masquerading as the operating system, the attacker can execute commands in a powerful role.
- Therefore, by replacing a few instructions right after returning from his or her own procedure, the attacker regains control from the operating system, possibly with raised privileges. This technique is called privilege escalation



Memory Organization with User and System Areas

Overwriting Memory

- A buffer (or array or string) is a space in which data can be held. A buffer resides in memory. Because memory is finite, a buffer's capacity is finite. For this reason, in many programming languages the programmer must declare the buffer's maximum size so that the compiler can set aside that amount of space.
- Example to see how buffer overflows can happen. Suppose a C language program contains the declaration

```
char sample[10];
```

The compiler sets aside 10 bytes to store this buffer, one byte for each of the 10 elements of the array, denoted sample[0] through sample[9]. Now we execute the statement

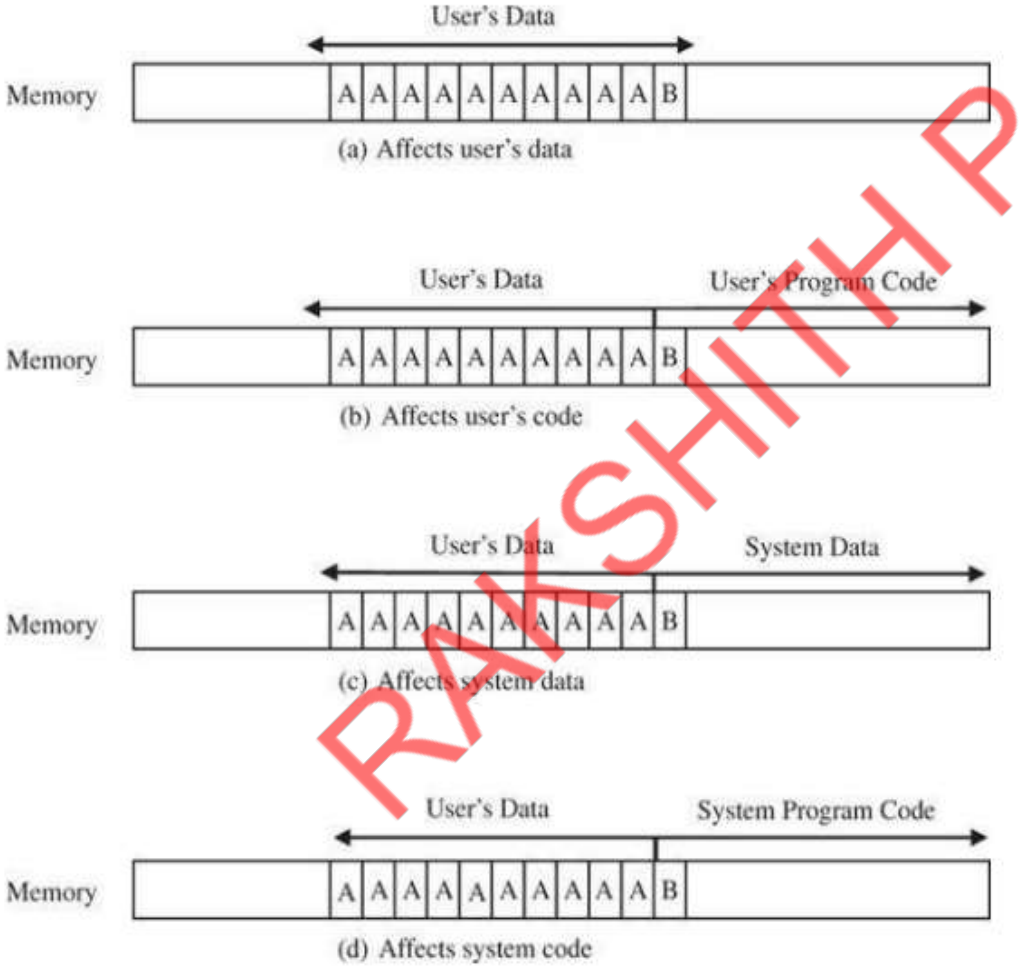
```
sample[10] = 'B';
```

Implications of Overwriting Memory

- Examine the problem of overwriting memory. Be sure to recognize that the potential overflow causes a serious problem only in some instances.
- The problem's occurrence depends on what is adjacent to the array sample. For example, suppose each of the ten elements of the array sample is filled with the letter A and the erroneous reference uses the letter B, as follows:


```
for (i=0; i<=9; i++)
        sample[i] = 'A';
sample[10] = 'B'
```

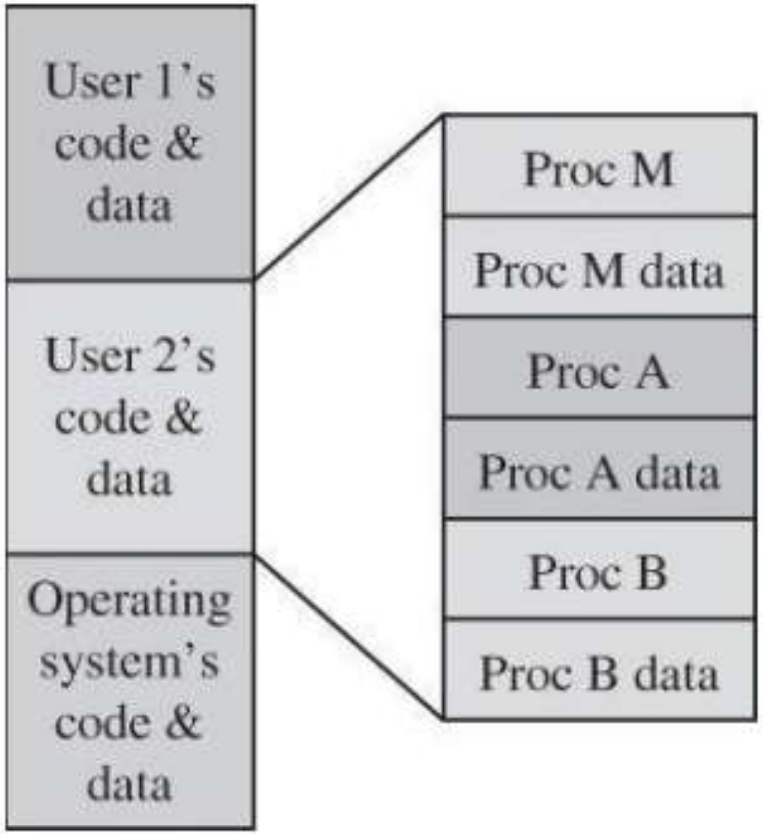
- All program and data elements are in memory during execution, sharing space with the operating system, other code, and resident routines. So four cases must be considered in deciding where the ‘B’ goes, as shown in Figure below.
- If the extra character overflows into the user’s data space, it simply overwrites an existing variable value (or it may be written into an as-yet unused location), perhaps affecting the program’s result but affecting no other program or data.



In the second case, the ‘B’ goes into the user’s program area. If it overlays an already executed instruction (which will not be executed again), the user should perceive no effect. If it overlays an instruction that is not yet executed, the machine will try to execute an instruction with operation code 0x42, the internal code for the character ‘B’.

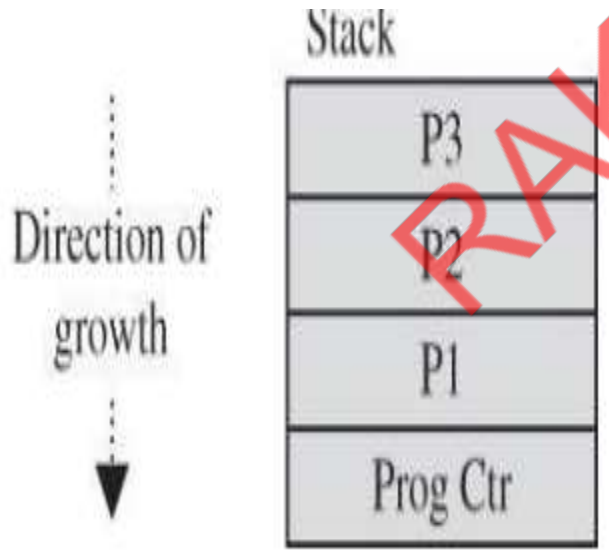
- Program procedures use both local data, data used strictly within one procedure, and shared or common or global data, which are shared between two or more procedures.
- Memory organization can be complicated, but we simplify the layout as in Figure below. In that picture, local data are stored adjacent to the code of a procedure. The data end up on top of one of

1. another piece of your data
2. an instruction of yours
3. data or code belonging to another program
4. data or code belonging to the operating system

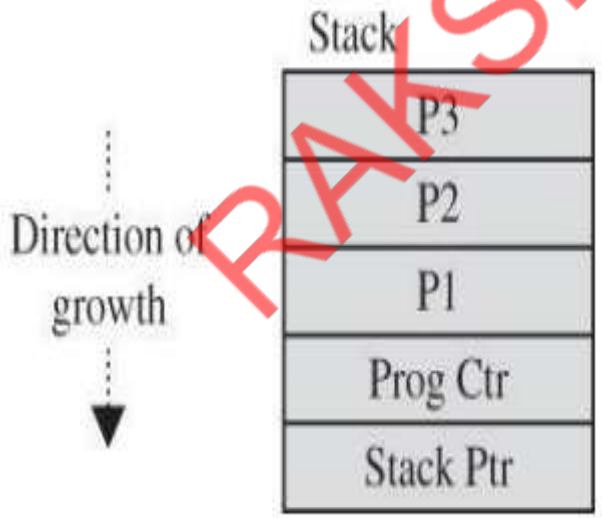


The Stack and the Heap

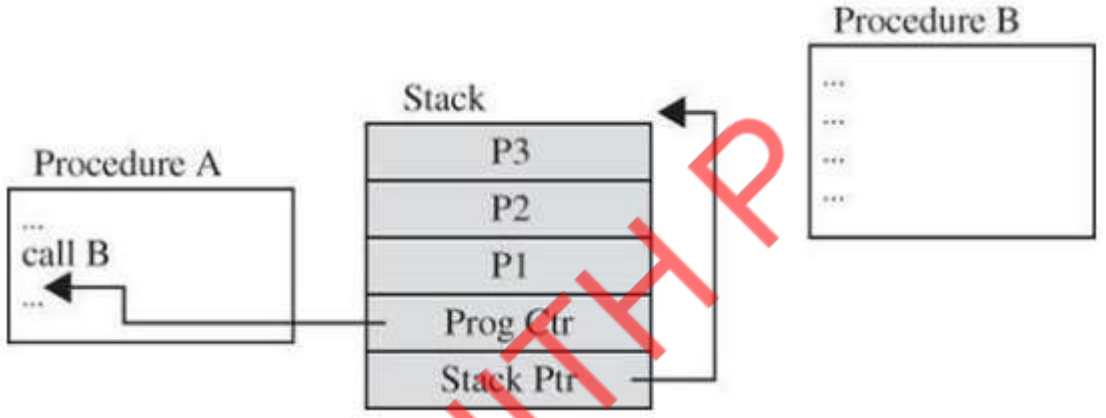
- The stack is a key data structure necessary for interchange of data between procedures.
- Executable code resides at one end of memory, which we depict as the low end; above it are constants and data items whose size is known at compile time; above that is the heap for data items whose size can change during execution; and finally, the stack.
- When procedure A calls procedure B, A pushes onto the stack its return address (that is, the current value of the program counter), the address at which execution should resume when B exits, as well as calling parameter values. Such a sequence is shown in Figure below.



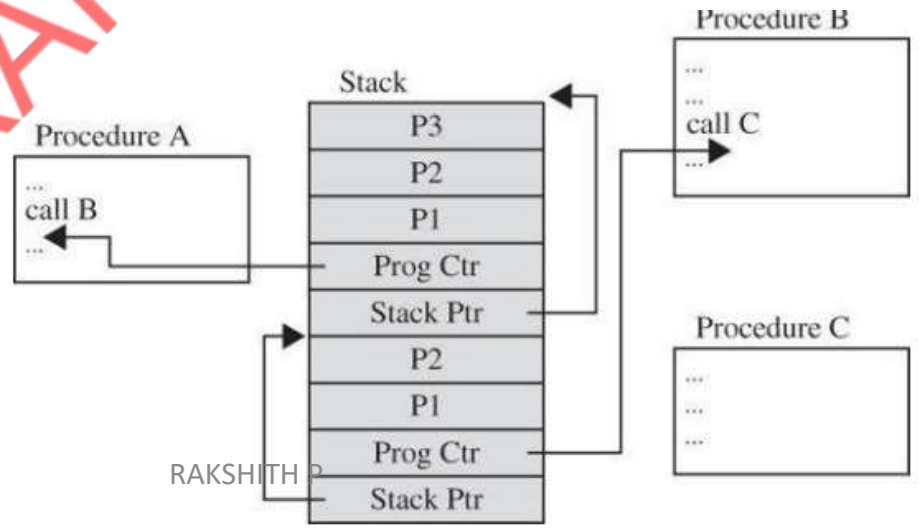
- To help unwind stack data tangled because of a program that fails during execution, the stack also contains the pointer to the logical bottom of this program's section of the stack, that is, to the point just before where this procedure pushed values onto the stack.
- This data group of parameters, return address, and stack pointer is called a stack frame, as shown in Figure below



- When one procedure calls another, the stack frame is pushed onto the stack to allow the two procedures to exchange data and transfer control; an example of the stack after procedure A calls B is shown in Figure below



- The return address to A on the bottom, then parameters from A to B, the return address from C to B, and parameters from B to C, in that order. After procedure C returns to B, the second stack frame is popped off the stack and it looks again like Figure below.



Code Analyzers

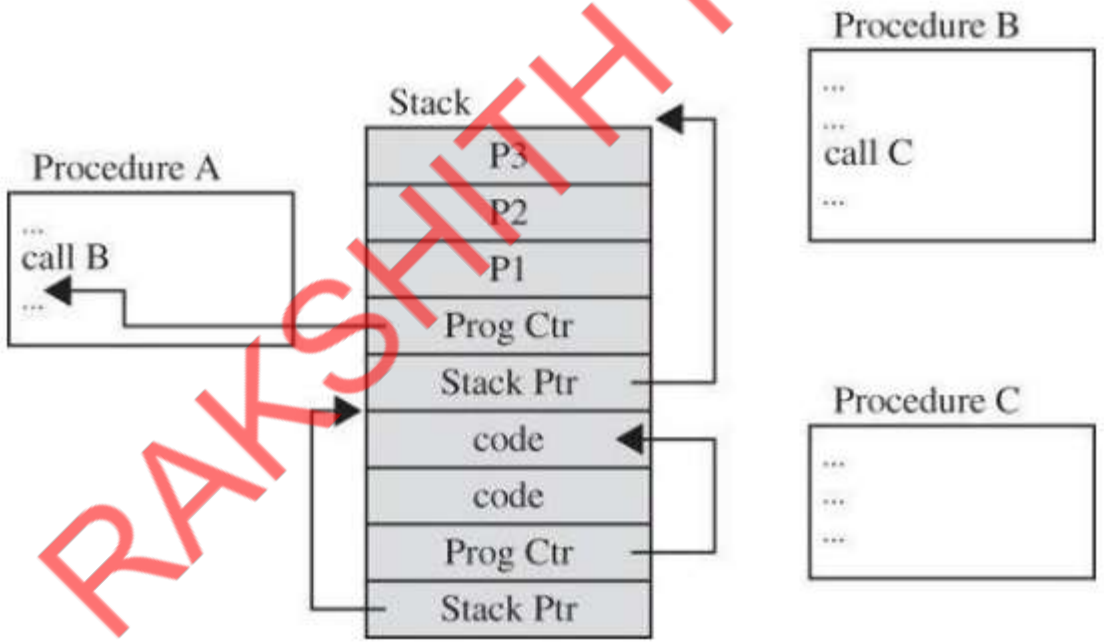
- Software developers hope for a simple tool to find security errors in programs.
- Such a tool, called a static code analyzer, analyzes source code to detect unsafe conditions. Although such tools are not, and can never be, perfect, several good ones exist

Separation

- Another direction for protecting against buffer overflows is to enforce containment: separating sensitive areas from the running code and its buffers and data space. To a certain degree, hardware can separate code from data areas and the operating system.

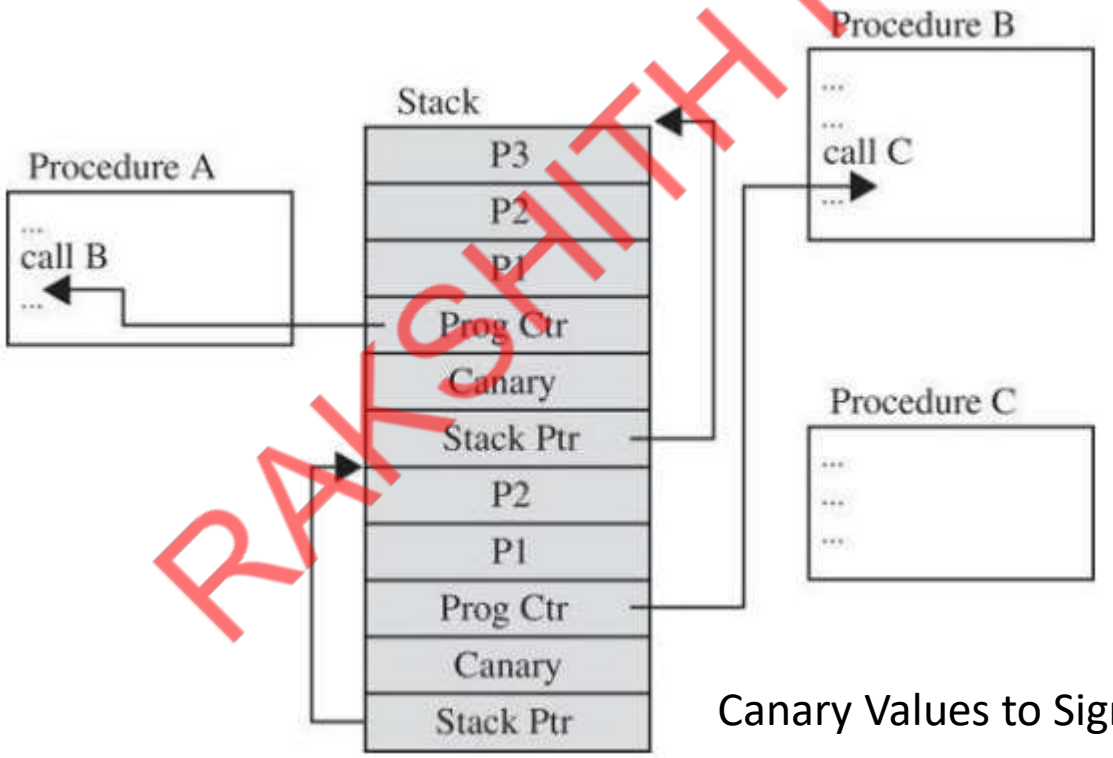
Stumbling Blocks

- In a common buffer overflow stack modification, the program counter is reset to point into the stack to the attack code that has overwritten stack data. In Figure below, the two parameters P1 and P2 have been overwritten with code to which the program counter has been redirected.



The attacker has to rewrite not just the stack pointer but also some words around it to be sure of changing the true stack pointer, but this uncertainty to the attacker allows StackGuard to detect likely changes to the program counter. Each procedure includes a prolog code to push values on the stack, set the remainder of the stack frame, and pass control to the called return; then on return, some termination code cleans up the stack, reloads registers, and returns.

- Each canary value serves as a protective insert to protect the program counter. These protective inserts are shown in Figure below.
- The idea of surrounding the return address with a tamper-detecting value is sound, as long as only the defender can generate and verify that value



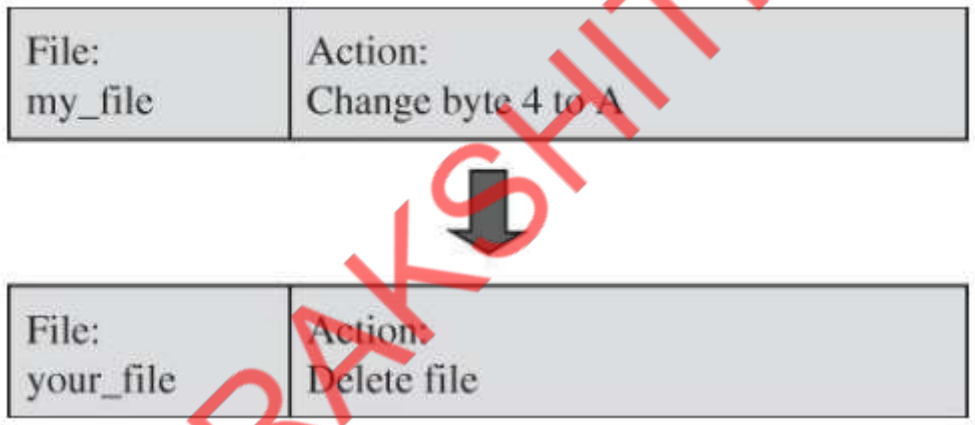
Canary Values to Signal Modification

Time-of-Check to Time-of-Use

- To improve efficiency, modern processors and operating systems usually change the order in which instructions and procedures are executed.
- In particular, instructions that appear to be adjacent may not actually be executed immediately after each other, either because of intentionally changed order or because of the effects of other processes in concurrent execution.
- Time-of-check-to-time-of-use (TOCTTOU - pronounced TOCK-too) is a file-based race condition that occurs when a resource is checked for a particular value, such as whether a file exists or not, and that value then changes before the resource is used, invalidating the results of the check.
- Suppose a request to access a file were presented as a data structure, with the name of the file and the mode of access presented in the structure. An example of such a structure is shown in Figure below.

File: my_file	Action: Change byte 4 to A

- At this point the incomplete mediation flaw can be exploited. While the mediator is checking access rights for the file my_file, the user could change the file name descriptor to your_file, the value shown in Figure below.
- Having read the work ticket once, the mediator would not be expected to reread the ticket before approving it; the mediator would approve the access and send the now-modified descriptor to the file handler.



Unchecked Change to Work Descriptor

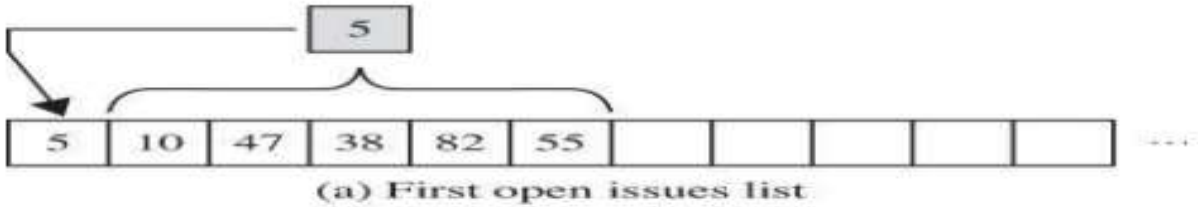
Backdoor

Secret backdoors are eventually found. Security cannot depend on such secrecy.

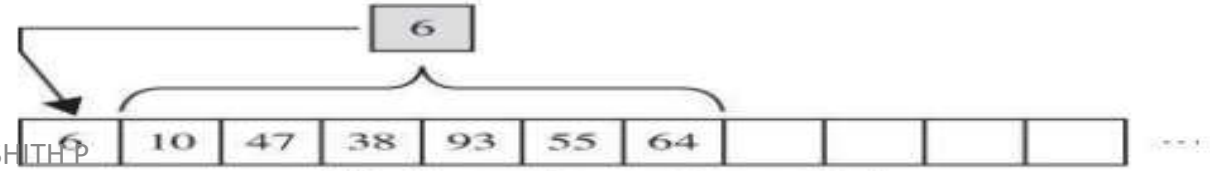
- An undocumented access point is called a backdoor or trapdoor. Such an entry can transfer control to any point with any privileges the programmer wanted.
- Few things remain secret on the web for long; someone finds an opening and exploits it. Thus, coding a supposedly secret entry point is an opening for unannounced visitors.
- A backdoor attack is a way to access a computer system or encrypted data that bypasses the system's customary security mechanisms. A developer may create a backdoor so that an application, operating system (OS) or data can be accessed for troubleshooting or other purposes

Off-by-One Error

- An off-by-one error or off-by-one bug (known by acronyms OBOE, OBO, OB1 and OBOB) is a logic error involving the discrete equivalent of a boundary condition. It often occurs in computer programming when an iterative loop iterates one time too many or too few.
- Off-by-one error: miscalculating the condition to end a loop (repeat while i< = n or in?) or overlooking that an array of A[0] through A[n] contains n+1 elements.
- For example, a program may manage a list that increases and decreases. Think of a list of unresolved problems in a customer service department: There are five open issues, numbered 10, 47, 38, 82, and 55; during the day, issue 82 is resolved but issues 93 and 64 are added to the list.
- A programmer may create a simple data structure, an array, to hold these issue numbers and may reasonably specify no more than 100 numbers.
- But to help with managing the numbers, the programmer may also reserve the first position in the array for the count of open issues.
- Thus, in the first case the array really holds six elements, 5 (the count), 10, 47, 38, 82, and 55; and in the second case there are seven, 6, 10, 47, 38, 93, 55, 64, as shown in Figure below. A 100-element array will clearly not hold 100 data items plus one count.



(a) First open issues list



(b) Second open issues list

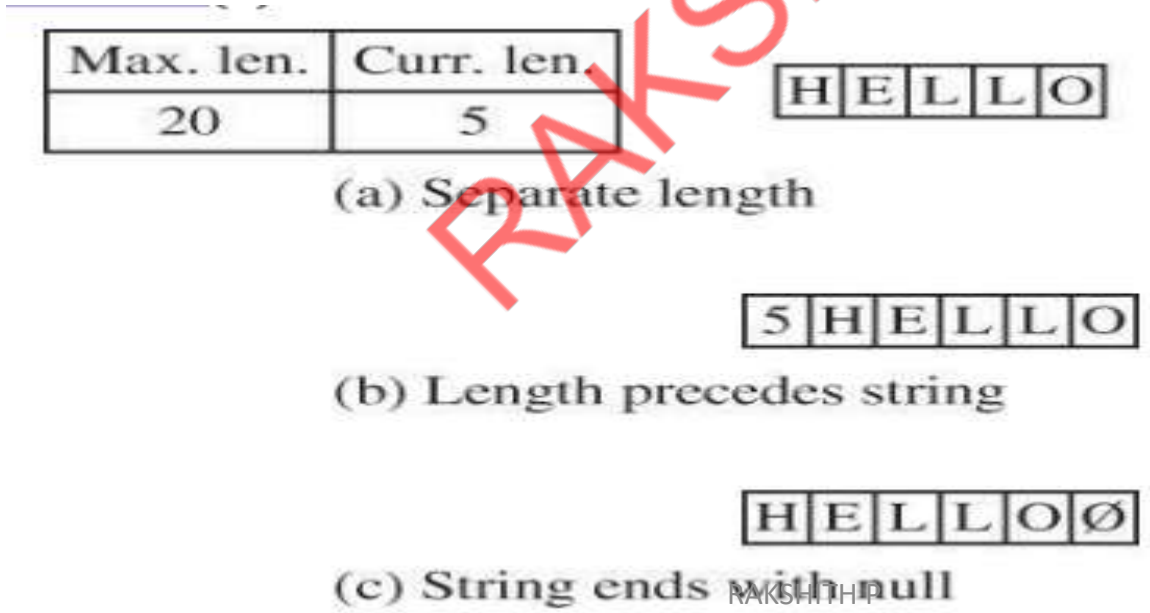
Integer Overflow

- An integer overflow occurs because a storage location is of fixed, finite size and therefore can contain only integers up to a certain limit.
- The overflow depends on whether the data values are signed (that is, whether one bit is reserved for indicating whether the number is positive or negative).
- Table gives the range of signed and unsigned values for several memory location (word) sizes

Word Size	Signed Values	Unsigned Values
8 bits	−128 to +127	0 to 255 ($2^8 - 1$)
16 bits	−32,768 to +32,767	0 to 65,535 ($2^{16} - 1$)
32 bits	−2,147,483,648 to +2,147,483,647	0 to 4,294,967,296 ($2^{32} - 1$)

Unterminated Null-Terminated String

- Long strings are the source of many buffer overflows. Sometimes an attacker intentionally feeds an overly long string into a processing program to see if and how the program will fail, as was true with the Dialer program.
- Other times the vulnerability has an accidental cause: A program mistakenly overwrites part of a string, causing the string to be interpreted as longer than it really is.



Parameter Length, Type, and Number

- Another source of data-length errors is procedure parameters, from web or conventional applications. Among the sources of problems are these:
- Too many parameters. Even though an application receives only three incoming parameters, for example, that application can incorrectly write four outgoing result parameters by using stray data adjacent to the legitimate parameters passed in the calling stack frame
- Wrong output type or size. A calling and called procedure need to agree on the type and size of data values exchanged. If the caller provides space for a two byte integer but the called routine produces a four-byte result, those extra two bytes will go somewhere.
- Too-long string. A procedure can receive as input a string longer than it can handle, or it can produce a too-long string on output, each of which will also cause an overflow condition.

Unsafe Utility Program

- Programming languages, especially C, provide a library of utility routines to assist with common activities, such as moving and copying strings. In C the function strcpy(dest, src) copies a string from src to dest, stopping on a null, with the potential to overrun allocated memory.
- A safer function is strncpy(dest, src, max), which copies up to the null delimiter or max characters, whichever comes first.

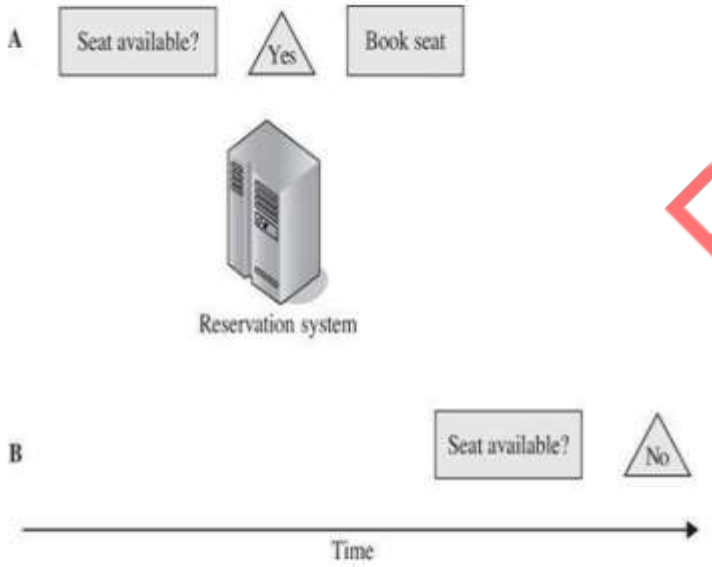
Race Condition

- As the name implies, a race condition means that two processes are competing within the same time interval, and the race affects the integrity or correctness of the computing tasks.
- For instance, two devices may submit competing requests to the operating system for a given chunk of memory at the same time.
- Device first asks if the size chunk is available, and if the answer is yes, then reserves that chunk for itself. Depending on the timing of the steps, the first device could ask for the chunk, get a “yes” answer, but then not get the chunk because it has already been assigned to the second device.
- A race condition occurs most often in an operating system, but it can also occur in multithreaded or cooperating processes.

- In a race condition or serialization flaw two processes execute concurrently, and the outcome of the computation depends on the order in which instructions of the processes execute

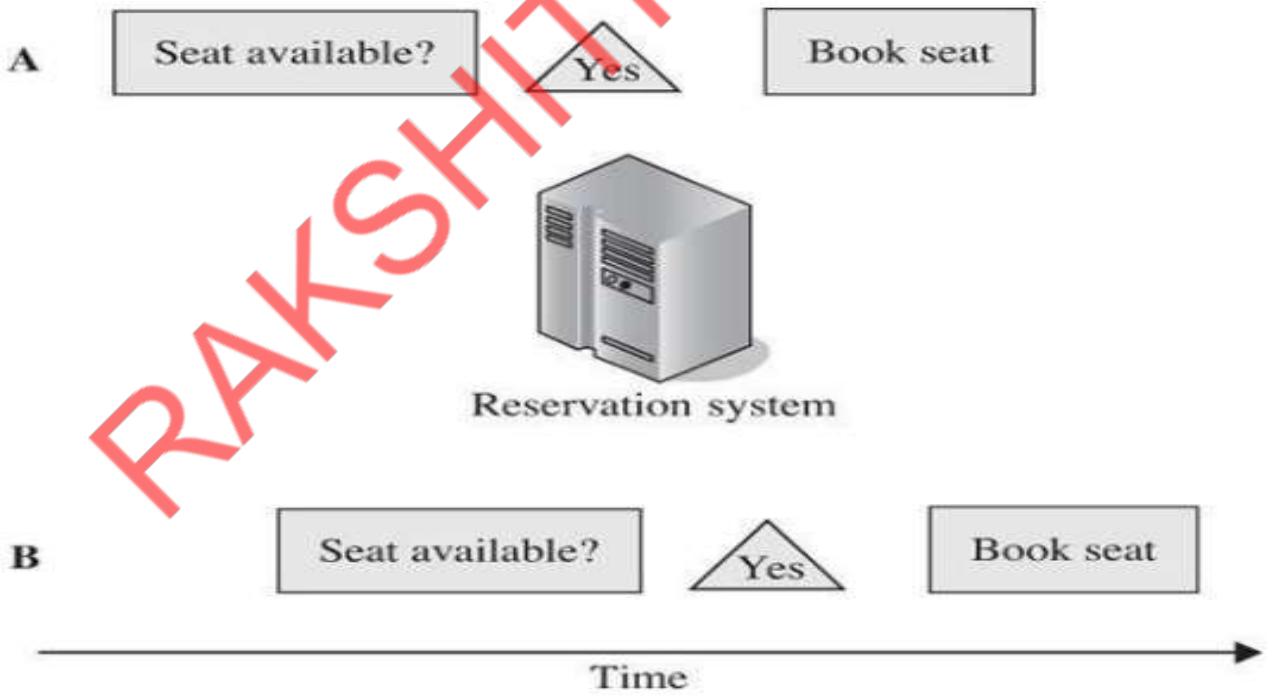
Race condition: situation in which program behavior depends on the order in which two procedures execute

Imagine an airline reservation system. Each of two agents, A and B, simultaneously tries to book a seat for a passenger on flight 45 on 10 January, for which there is exactly one seat available. If agent A completes the booking before that for B begins, A gets the seat and B is informed that no seats are available. In Figure 3-16 we show a timeline for this situation.



Seat Request and Reservation Example

- Imagine a situation in which A asks if a seat is available, is told yes, and proceeds to complete the purchase of that seat.
- Meanwhile, between the time A asks and then tries to complete the purchase, agent B asks if a seat is available. The system designers knew that sometimes agents inquire about seats but never complete the booking; their clients often choose different itineraries once they explore their options
- A has not completed the transaction before the system gets a request from B, the system tells B that the seat is available. If the system is not designed properly, both agents can complete their transactions, and two passengers will be confirmed for that one seat (which will be uncomfortable, to say the least). We show this timeline in Figure below



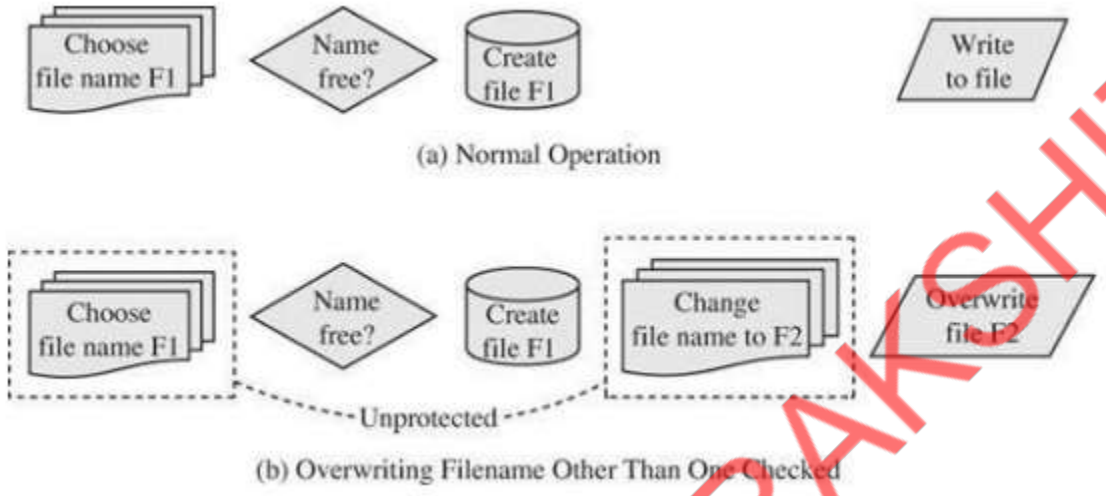
Overbooking Example

- A race condition is difficult to detect because it depends on the order in which two processes execute. But the execution order of the processes can depend on many other things, such as the total load on the system, the amount of available memory space, the priority of each process, or the number and timing of system interrupts to the processes.
- During testing, and even for a long period of execution, conditions may never cause this particular overload condition to occur.
- Most of today's computers are configured with applications selected by their owners, tailored specifically for the owner's activities and needs.
- These applications, as well as the operating system and device drivers, are likely to be produced by different vendors with different design strategies, development philosophies, and testing protocols.
- The likelihood of a race condition increases with this increasing system heterogeneity.

Security Implication

- The security implication of race conditions is evident from the airline reservation example.
- A race condition between two processes can cause inconsistent, undesired and therefore wrong, outcomes—a failure of integrity.
- As part of its operation it creates a temporary file to which it writes a log of its activity. In the old version, Tripwire
- (1) chose a name for the temporary file,
- (2) checked the file system to ensure that no file of that name already existed,
- (3) created a file by that name, and
- (4) later opened the file and wrote results

- Any file can be compromised by a carefully timed use of the inherent race condition between steps 2 and 3, as shown in Figure below.
- Overwriting a file may seem rather futile or self-destructive, but an attacker gains a strong benefit



File Name Race Condition

Race conditions depend on the order and timing of two different processes, making these errors hard to find (and test for).

If the malicious programmer acts too early, no temporary file has yet been created, and if the programmer acts too late, the file has been created and is already in use.

But if the programmer's timing is between too early and too late, Tripwire will innocently write its temporary data over whatever file is pointed at.

Although this timing may seem to be a serious constraint, the attacker has an advantage: If the attacker is too early, the attacker can try again and again until either the attack succeeds or is too late.

- Thus, race conditions can be hard to detect; testers are challenged to set up exactly the necessary conditions of system load and timing.
- For the same reason, race condition threats are hard for the attacker to execute. Nevertheless, if race condition vulnerabilities exist, they can also be exploited.
- The vulnerabilities presented here—incomplete mediation, race conditions, time-of-check to time-of-use, and undocumented access points—are flaws that can be exploited to cause a failure of security.
- Throughout this we describe other sources of failures because programmers have many process points to exploit and opportunities to create program flaws.

Malicious Code—Malware:

- In May 2010, researcher Roger Thompson of the antivirus firm AVG detected malicious code at the web site of the U.S. Bureau of Engraving and Printing, a part of the Treasury Department [MCM10].
- The source of the exploit is unknown; some researchers think it was slipped into the site's tracking tool that tallies and displays the number of visits to a web page.
- As malicious code attacks go, this one was not the most sophisticated, complicated, or devastating.

Malware—Viruses, Trojan Horses, and Worms

- Malicious code or rogue programs or malware (short for MALicious softWARE) is the general name for programs or program parts planted by an agent with malicious intent to cause unanticipated or undesired effects. The agent is the program's writer or distributor.
- Malicious intent distinguishes this type of code from unintentional errors, even though both kinds can certainly have similar and serious negative effects.
- Most faults found in software inspections, reviews, and testing do not qualify as malicious code; their cause is usually unintentional.
- However, unintentional faults can in fact invoke the same responses as intentional malevolence; a benign cause can still lead to a disastrous effect.

Malicious code can be directed at a specific user or class of users, or it can be for anyone.

- The malware may have been caused by a worm or a virus or neither; the infection metaphor often seems apt, but the terminology of malicious code is sometimes used imprecisely.
- A virus is a program that can replicate itself and pass on malicious code to other nonmalicious programs by modifying them.
- The term “virus” was coined because the affected program acts like a biological virus: It infects other healthy subjects by attaching itself to the program and either destroying the program or coexisting with it.
- viruses are insidious, we cannot assume that a clean program yesterday is still clean today. Moreover, a good program can be modified to include a copy of the virus program, so the infected good program itself begins to act as a virus, infecting other programs.
- The infection usually spreads at a geometric rate, eventually overtaking an entire computing system and spreading to other connected systems.

Virus: code with malicious purpose; intended to spread

- A virus can be either transient or resident. A transient virus has a life span that depends on the life of its host; the virus runs when the program to which it is attached executes, and it terminates when the attached program ends.

- A resident virus locates itself in memory; it can then remain active or be activated as a stand-alone program, even after its attached program ends.
- The terms worm and virus are often used interchangeably, but they actually refer to different things.
- A worm is a program that spreads copies of itself through a network.

Malware doesn't attack just individual users and single computers.
Major applications and industries are also at risk.

Worm: program that spreads copies of itself through a network

- A bot (short for robot), is a kind of worm used in vast numbers by search engine hosts like Bing and Google.
- A Trojan horse is malicious code that, in addition to its primary effect, has a second, nonobvious, malicious effect.
- Trojan horse, consider a login script that solicits a user's identification and password, passes the identification information on to the rest of the system for login processing, but also retains a copy of the information for later, malicious use.

Trojan horse: program with benign apparent effect but second, hidden, malicious effect

Code Type	Characteristics
Virus	Code that causes malicious behavior and propagates copies of itself to other programs
Trojan horse	Code that contains unexpected, undocumented, additional functionality
Worm	Code that propagates copies of itself through a network; impact is usually degraded performance
Rabbit	Code that replicates itself without limit to exhaust resources
Logic bomb	Code that triggers action when a predetermined condition occurs
Time bomb	Code that triggers action when a predetermined time occurs
Dropper	Transfer agent code only to drop other malicious code, such as virus or Trojan horse
Hostile mobile code agent	Code communicated semi-autonomously by programs transmitted through the web
Script attack, JavaScript, Active code attack	Malicious code communicated in JavaScript, ActiveX, or another scripting language, downloaded as part of displaying a web page
RAT (remote access Trojan)	Trojan horse that, once planted, gives access from remote location
Spyware	Program that intercepts and covertly communicates data on the user or the user's activity
Bot	Semi-autonomous agent, under control of a (usually remote) controller or "herder"; not necessarily malicious
Zombie	Code or entire computer under control of a (usually remote) program
Browser hijacker	Code that changes browser settings, disallows access to certain sites, or redirects browser to others
Rootkit	Code installed in "root" or most privileged section of operating system; hard to detect
Trapdoor or backdoor	Code feature that allows unauthorized access to a machine or program; bypasses normal access control and authentication
Tool or toolkit	Program containing a set of tests for vulnerabilities; not dangerous itself, but each successful test identifies a vulnerable host that can be attacked
Scareware	Not code; false warning of malicious code attack

Types of Malicious Code

- we explore four aspects of malicious code infections:
- • Harm—how they affect users and systems
- • Transmission and propagation—how they are transmitted and replicate, and how they cause further transmission
- • Activation—how they gain control and install themselves so that they can reactivate
- • Stealth—how they hide to avoid detection

Harm from Malicious Code

- Viruses and other malicious code can cause essentially unlimited harm. Because malware runs under the authority of the user, it can do anything the user can do.
- We can divide the payload from malicious code into three categories:
- Nondestructive. Examples of behavior are sending a funny message or flashing an image on the screen, often simply to show the author's capability
- Destructive. This type of code corrupts files, deletes files, damages software, or executes commands to cause hardware stress or breakage with no apparent motive other than to harm the recipient.
- Commercial or criminal intent. An infection of this type tries to take over the recipient's computer, installing code to allow a remote agent to cause the computer to perform actions on the agent's signal or to forward sensitive data to the agent.

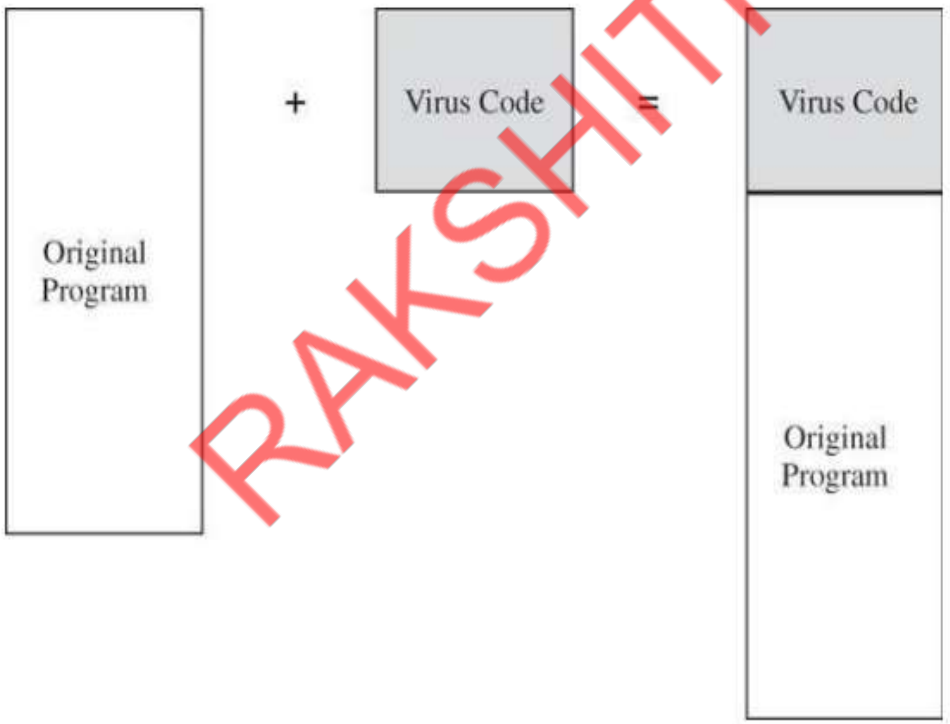
Harm to Users

Most malicious code harm occurs to the infected computer's data. Here are some real-world examples of malice.

- • Hiding the cursor.
- • Displaying text or an image on the screen.
- • Opening a browser window to web sites related to current activity (for example, opening an airline web page when the current site is a foreign city's tourist board).
- • Sending email to some or all entries in the user's contacts or alias list. Note that the email would be delivered as having come from the user, leading the recipient to think it authentic.
- • Opening text documents and changing some instances of "is" to "is not," and vice versa.
- Deleting all files.
- • Modifying system program files.
- • Stealing and forwarding sensitive information such as passwords and login details.
- • Hide copies of the executable code in more than one location.
- • Hide copies of the executable in different locations on different systems so no single eradication procedure can work.
- • Modify the system registry so that the malware is always executed or malware detection is disabled.

Appended Viruses

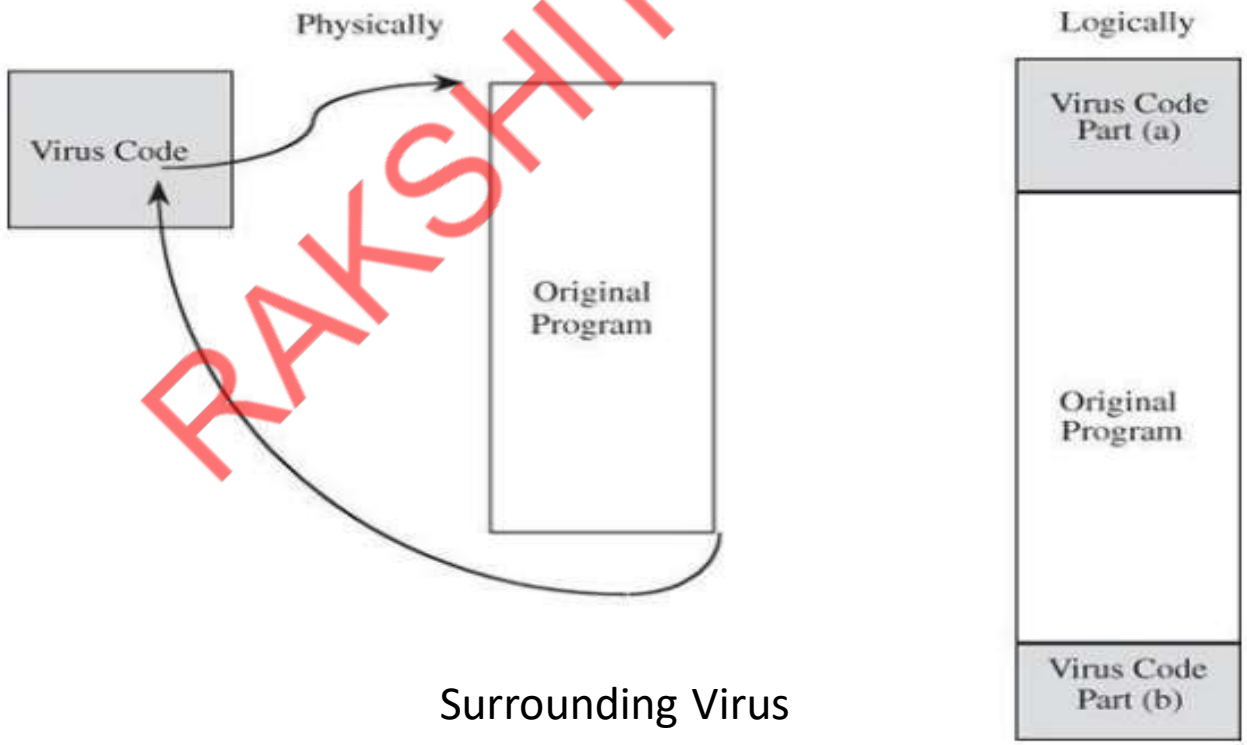
- A program virus attaches itself to a program; then, whenever the program is run, the virus is activated. This kind of attachment is usually easy to design and implement.
- In the simplest case, a virus inserts a copy of itself into the executable program file before the first executable instruction.
- Then, all the virus instructions execute first; after the last virus instruction, control flows naturally to what used to be the first program instruction. Such a situation is shown in Figure



Virus Attachment

Viruses That Surround a Program

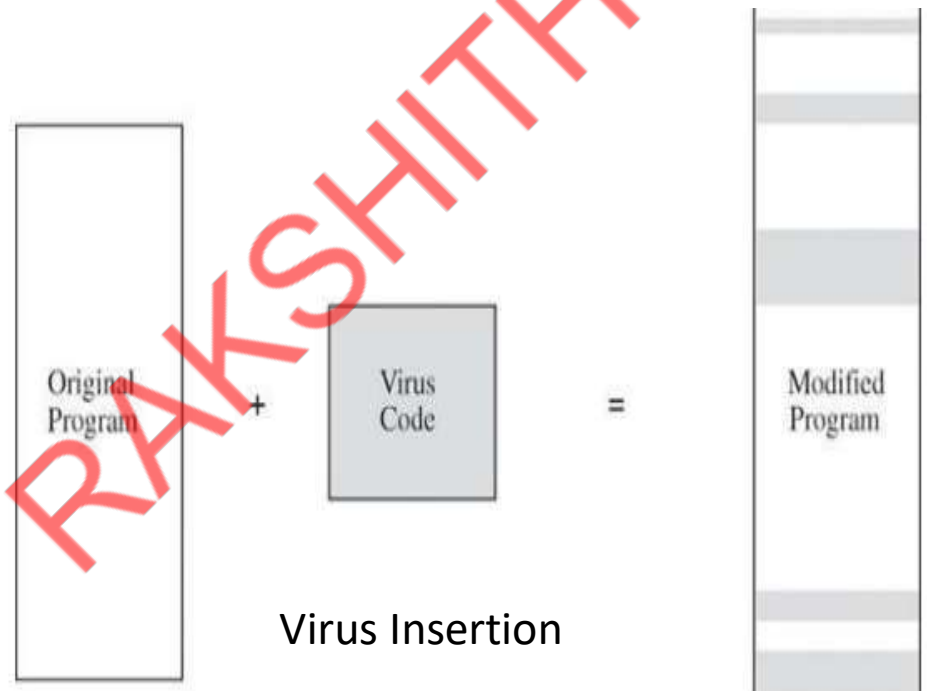
- An alternative to the attachment is a virus that runs the original program but has control before and after its execution.
- For example, a virus writer might want to prevent the virus from being detected. If the virus is stored on disk, its presence will be given away by its file name, or its size will affect the amount of space used on the disk.
- The virus writer might arrange for the virus to attach itself to the program that constructs the listing of files on the disk.
- If the virus regains control after the listing program has generated the listing but before the listing is displayed or printed, the virus could eliminate its entry from the listing and falsify space counts so that it appears not to exist. A surrounding virus is shown in Figure



Surrounding Virus

Integrated Viruses and Replacements

- A third situation occurs when the virus replaces some of its target, integrating itself into the original code of the target. Such a situation is shown in Figure . Clearly, the virus writer has to know the exact structure of the original program to know where to insert which pieces of the virus.

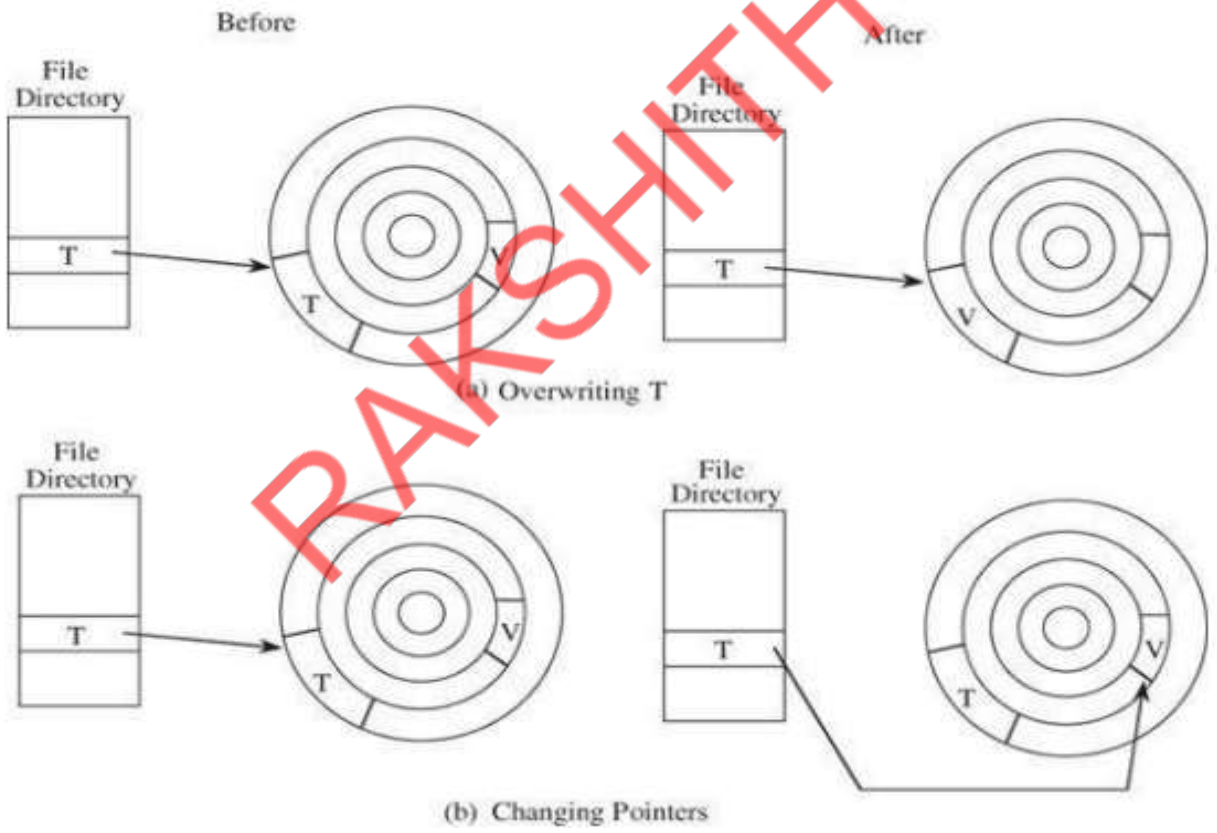


Virus Insertion

Finally, the malicious code can replace an entire target, either mimicking the effect of the target or ignoring its expected effect and performing only the virus effect.

How Malicious Code Gains Control

- To gain control of processing, malicious code such as a virus (V) has to be invoked instead of the target (T). Essentially, the virus either has to seem to be T, saying effectively "I am T," or the virus has to push T out of the way and become a substitute for T, saying effectively "Call me instead of T."
- A more blatant virus can simply say "invoke me [you fool]." The virus can assume T's name by replacing (or joining to) T's code in a file structure; this invocation technique is most appropriate for ordinary programs.
- The virus can overwrite T in storage (simply replacing the copy of T in storage, for example).
- Alternatively, the virus can change the pointers in the file table so that the virus is located instead of T whenever T is accessed through the file system. These two cases are shown in Figure

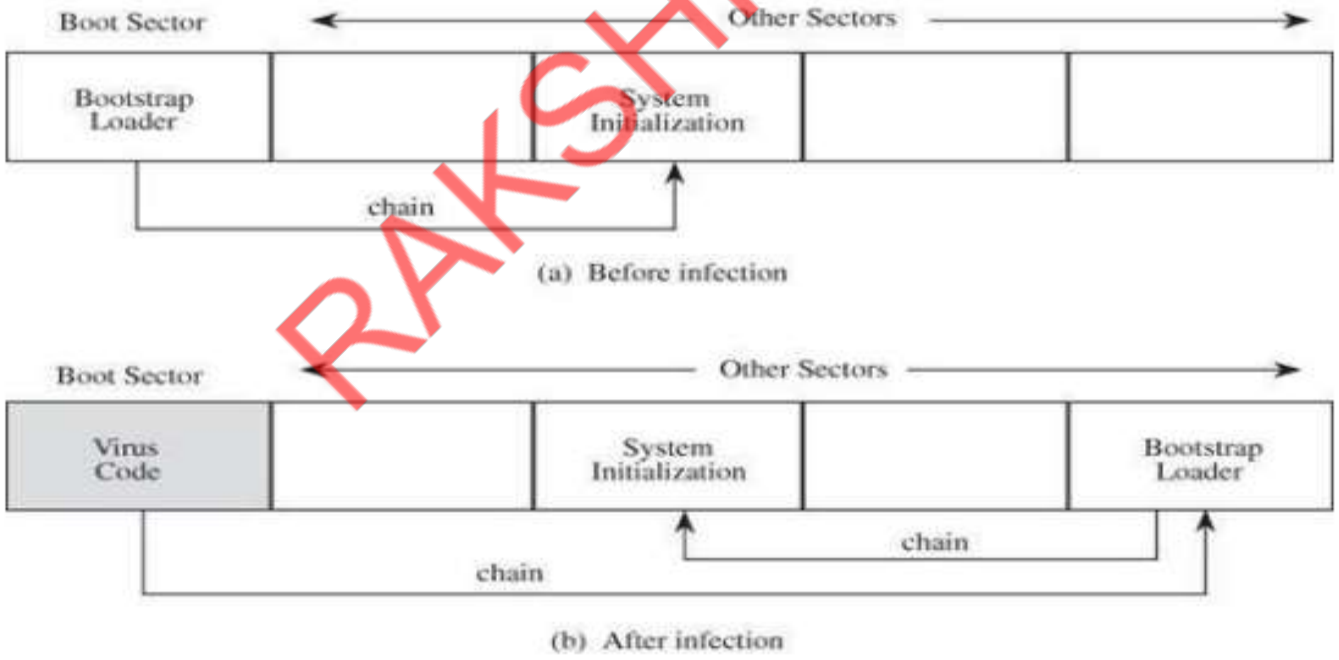


Virus V Replacing Target T

Boot Sector Viruses

- A special case of virus attachment, but formerly a fairly popular one, is the so-called boot sector virus.
- Attackers are interested in creating continuing or repeated harm, instead of just a one-time assault. For continuity the infection needs to stay around and become an integral part of the operating system.
- In such attackers, the easy way to become permanent is to force the harmful code to be reloaded each time the system is restarted.
- When a computer is started, control begins with firmware that determines which hardware components are present, tests them, and transfers control to an operating system.
- A given hardware platform can run many different operating systems, so the operating system is not coded in firmware but is instead invoked dynamically, perhaps even by a user's choice, after the hardware test.
- An executive oversees the boot process, loading and initiating the right modules in an acceptable order. Putting together a jigsaw puzzle is hard enough, but the executive has to work with pieces from many puzzles at once, somehow putting together just a few pieces from each to form a consistent, connected whole, without even a picture of what the result will look like when it is assembled.

- Malicious code can intrude in this bootstrap sequence in several ways. An assault can revise or add to the list of modules to be loaded, or substitute an infected module for a good one by changing the address of the module to be loaded or by substituting a modified routine of the same name.
- With boot sector attacks, the assailant changes the pointer to the next part of the operating system to load, as shown in Figure



Memory-Resident Viruses

- Some parts of the operating system and most user programs execute, terminate, and disappear, with their space in memory then being available for anything executed later.
- For frequently used parts of the operating system and for a few specialized user programs, it would take too long to reload the program each time it is needed.
- Instead, such code remains in memory and is called “resident” code. Examples of resident code are the routine that interprets keys pressed on the keyboard, the code that handles error conditions that arise during a program’s execution, or a program that acts like an alarm clock, sounding a signal at a time the user determines.
- Resident routines are sometimes called TSRs or “terminate and stay resident” routines

- A virus can also modify the operating system's table of programs to run. Once the virus gains control, it can insert a registry entry so that it will be reinvoked each time the system restarts.

- In this way, even if the user notices and deletes the executing copy of the virus from memory, the system will resurrect the virus on the next system restart.

- For general malware, executing just once from memory has the obvious disadvantage of only one opportunity to cause malicious behavior, but on the other hand, if the infectious code disappears whenever the machine is shut down, the malicious code is less likely to be analyzed by security teams.

Stealth

- The final objective for a malicious code writer is stealth: avoiding detection during installation, while executing, or even at rest in storage.

Most viruses maintain stealth by concealing their action, not announcing their presence, and disguising their appearance.

Detection

Malicious code discovery could be aided with a procedure to determine if two programs are equivalent: We could write a program with a known harmful effect, and then compare with any other suspect program to determine if the two have equivalent results. However, this equivalence problem is complex, and theoretical results in computing suggest that a general solution is unlikely.

Installation Stealth

- Several approaches used to transmit code without the user's being aware, including downloading as a result of loading a web page and advertising one function while implementing another.
- Malicious code designers are fairly competent at tricking the user into accepting malware.

• Execution Stealth

Modern operating systems often support dozens of concurrent processes, many of which have unrecognizable names and functions.

Thus, even if a user does notice a program with an unrecognized name, the user is more likely to accept it as a system program than malware.

Stealth in Storage

- Designing your code this way is the economical approach for you: Designing, coding, testing, and maintaining one entity for many customers is less expensive than doing that for each individual sale.
- Your delivered and installed code will then have sections of identical instructions across all copies.
- Antivirus and other malicious code scanners look for patterns because malware writers have the same considerations you would have in developing mass-market software:
- They want to write one body of code and distribute it to all their victims. That identical code becomes a pattern on disk for which a scanner can search quickly and efficiently

Execution Patterns

- A virus writer may want a virus to do several things at the same time, namely, spread infection, avoid detection, and cause harm.
- These goals are shown in Table below, along with ways each goal can be addressed. Unfortunately, many of these behaviors are perfectly normal and might otherwise go undetected.
- For instance, one goal is modifying the file directory; many normal programs create files, delete files, and write to storage media. Thus, no key signals point to the presence of a virus.

Virus Effect	How It Is Caused
Attach to executable program	• Modify file directory • Write to executable program file
Attach to data or control file	• Modify directory • Rewrite data • Append to data • Append data to self
Remain in memory	• Intercept interrupt by modifying interrupt handler address table • Load self in nontransient memory area
Infect disks	• Intercept interrupt • Intercept operating system call (to format disk, for example) • Modify system file • Modify ordinary executable program
Conceal self	• Intercept system calls that would reveal self and falsify result • Classify self as "hidden" file
Spread infection	• Infect boot sector • Infect system program • Infect ordinary program • Infect data ordinary program reads to control its execution
Prevent deactivation	• Activate before deactivating program and block deactivation • Store copy to reinfect after deactivation

Virus Effects and What They Cause

Transmission Patterns

- A virus is effective only if it has some means of transmission from one location to another.
- Viruses can travel during the boot process by attaching to an executable file or traveling within data files.
- The travel itself occurs during execution of an already infected program.
- Since a virus can execute any instructions a program can, virus travel is not confined to any single medium or execution pattern

• Polymorphic Viruses

The virus signature may be the most reliable way for a virus scanner to identify a virus.

If a particular virus always begins with the string 0x47F0F00E08 and has string 0x00113FFF located at word 12, other programs or data files are not likely to have these exact characteristics. For longer signatures, the probability of a correct match increases.

For example, the virus could have two alternative but equivalent beginning words; after being installed, the virus will choose one of the two words for its initial word.

Then, a virus scanner would have to look for both patterns. A virus that can change its appearance is called a polymorphic virus. (Poly means “many” and morph means “form.”) .

A simple variety of polymorphic virus uses encryption under various keys to make the stored form of the virus different. These are sometimes called encrypting viruses. This type of virus must contain three distinct parts: a decryption key, the (encrypted) object code of the virus, and the (unencrypted) object code of the decryption routine

Malware Toolkits

Malware toolkits let novice attackers probe for many vulnerabilities at the press of a button.

- Malicious Tools are malicious software programs that have been designed for automatically creating viruses, worms or Trojans , conducting DoS attacks on remote servers, hacking other computers, and more.
- Computer attacking is somewhat different. First, there is a thriving underground of web sites for hackers to exchange techniques and knowledge. (As with any web site, the reader has to assess the quality of the content.)
- Second, attackers can often experiment in their own laboratories (homes) before launching public strikes. Most importantly, malware toolkits are readily available for sale.