Programs and Programming

ICT 3156

Introduction

- Programs are simple things but they can wield mighty power.
- Programs are just strings of 0s and 1s, representing elementary machine commands.
- Security failures can result from **intentional** or **non-malicious** causes; both can cause harm.

Fault, Failure, and Flaw

Every failure has at least one fault as its root cause. But not every fault corresponds to a failure.

Fault

- When a human makes a mistake, called an **error**, in performing some software activity, the error may lead to a fault.
- **Fault**: **An incorrect step**, command, process, or data definition in a computer program, design, or documentation.
- A fault is an inside view of the system, as seen by the eyes of the developers.

Failure

- A **failure** is a departure from the system's required behavior.
- It can be discovered before or after system delivery, during testing, or during operation and maintenance.
- A failure is an outside view: a problem that the user sees.

Program Flaws

- Program flaws can have two kinds of security implications:

1. They can cause integrity problems leading to harmful output or action.
    - Integrity involves correctness, accuracy, precision, and consistency.
    - A faulty program can also inappropriately modify previously correct data, sometimes by overwriting or deleting the original data.
2. They offer an opportunity for exploitation by a malicious actor.

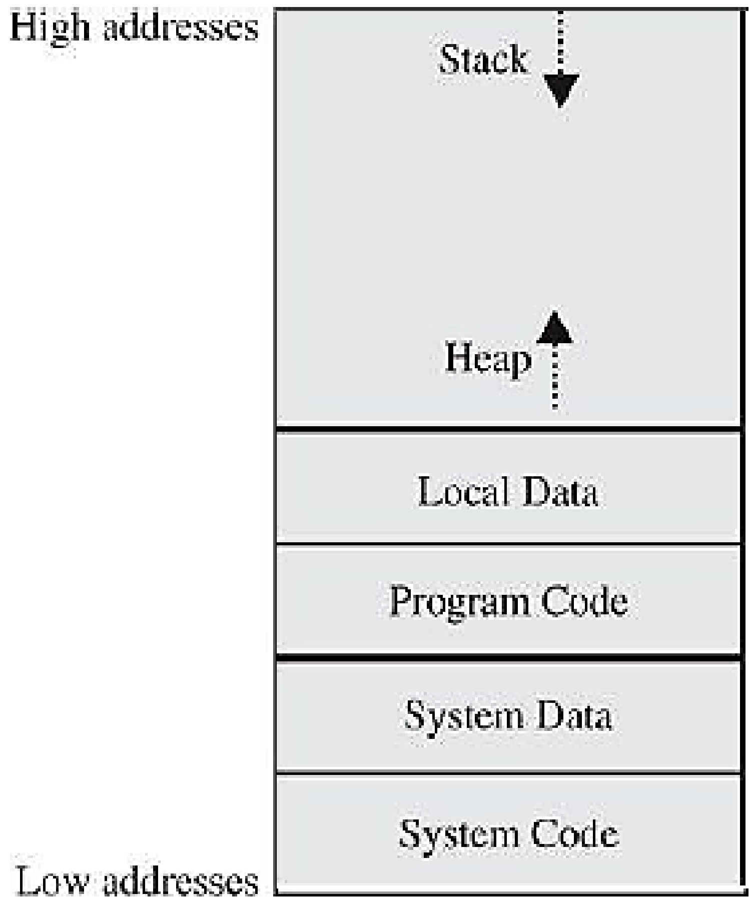
- **Programming Oversights**
    - Buffer Overflows
    - Off-by-one Errors
    - Incomplete Mediation
    - Time-of-check To Time-of-use Errors

Buffer Overflows

- A buffer (or array or string) is a space in which data can be held.
- Because memory is finite, a buffer's capacity is finite. Static and dynamic allocation.
- Buffer overflow: when user input exceeds max buffer size.
- Buffer overflows often come from innocent programmer oversights or failures to document and check for excessive data.
- Buffer overflow attacks are examples of **data driven attack;** here the harm occurs by the data the attacker sends.
- Case Study: David Litchfield, 1999.
- To understand buffer overflow, we need to first understand how memory is allocated.

Buffer Overflow

- In memory, code is indistinguishable from data. The origin of code (respected source or attacker) is also not visible.
- Any memory location can hold any piece of code or data.
- Computers use a pointer or register known as a **program counter** that indicates the next instruction.
- Usually we do not treat code as data, or vice versa. However, attackers sometimes do so.
- The attacker's trick is to **cause data to spill** over into executable code and then to select the data values such that they are **interpreted as valid instructions to perform the attacker's goal.**



Harm from Overflow

- The operating system’s code and data coexist with a user’s code and data.
- The attacker may replace code in the system space.

- Every program is invoked by an operating system that may run with higher privileges than those of a regular program.
- If the attacker can gain control by masquerading as the operating system, the attacker can execute commands in a powerful role. This technique is called **privilege escalation**.

Harm from Overflow

- The intruder may wander into an area called the stack and heap.
- The stack and heap grow toward each other, and you can predict that at some point they might collide.
- **Stack Smashing:** The attacker wants to overwrite stack memory, in a purposeful manner: Arbitrary data in the wrong place causes strange behavior, but particular data in a predictable location causes a planned impact.
- Some ways the attacker can produce effects from an overflow attack:

1. Overwrite the program counter.
2. Overwrite part of the code in low memory.
3. Overwrite the program counter and data in the stack.

Common feature of these attack methods?

Buffer Overflow: Example

```
char sample[10];
1.
sample[10] = ‘B’;
```

The subscript is out of bounds.

```
2.
sample[i] = ‘B’;
```

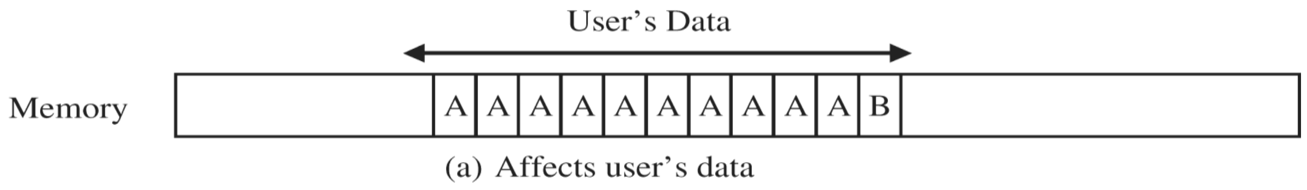
Buffer Overflow: Example

```
char sample[10];
1.
sample[10] = ‘B’;
```

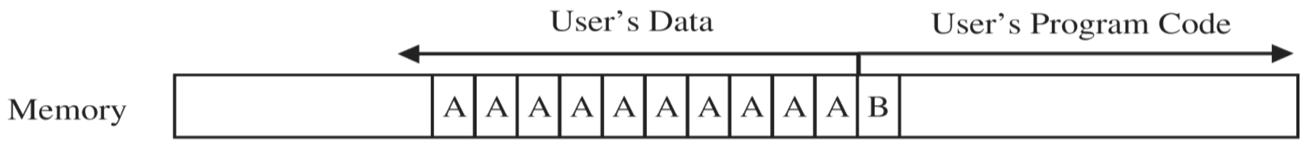
The subscript is out of bounds.

```
2.
sample[i] = ‘B’;

3.
for (i=0; i<=9; i++)
   sample[i] = 'A’;
sample[10] = 'B'
```



(a) Affects user’s data



(b) Affects user’s code

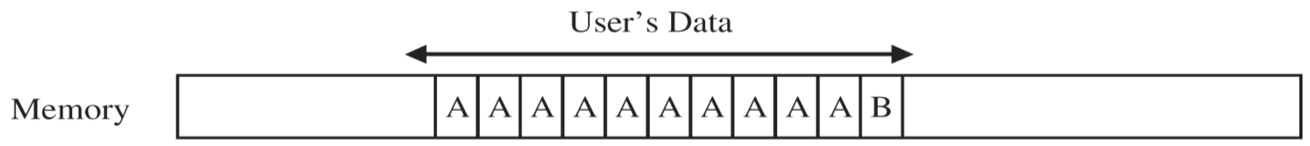
Buffer Overflow: Example

```
char sample[10];
1.
sample[10] = ‘B’;
```

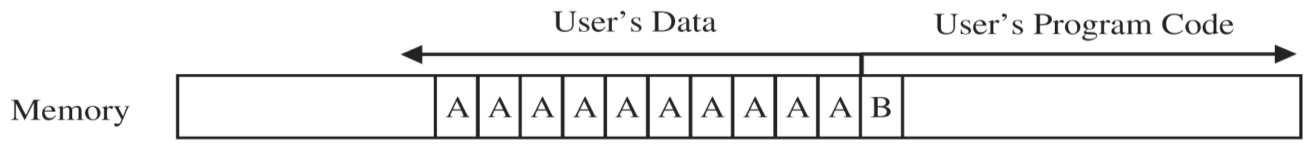
The subscript is out of bounds.

```
2.
sample[i] = ‘B’;

3.
for (i=0; i<=9; i++)
   sample[i] = 'A’;
sample[10] = 'B'
```

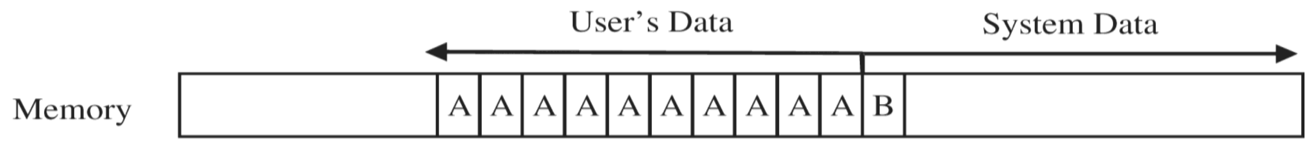


(a) Affects user’s data

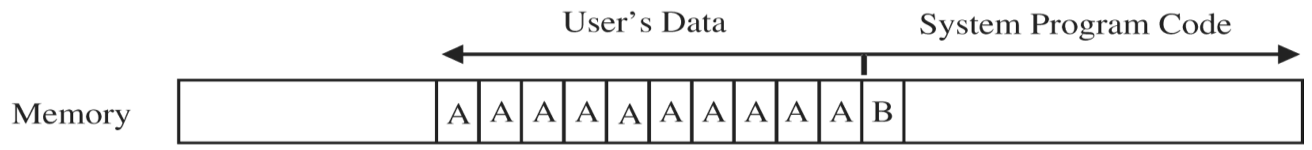


(b) Affects user’s code

Affecting Your Own Data and Instruction.



(c) Affects system data



(d) Affects system code

Overflow Countermeasures

- The most obvious countermeasure to overwriting memory is to stay within bounds.
- Maintaining boundaries is a shared responsibility of the programmer, operating system, compiler, and hardware.
- Check **lengths** before writing.
- Confirm that array **subscripts** are within **limits**.
- Double-check **boundary condition code** to catch possible off-by-one errors.
- Monitor **input** and accept only as many characters as can be handled.
- Use string **utilities** that transfer only a **bounded** amount of data.
- Check procedures that might **overrun** their space.
- Limit programs' **privileges**.

Off-by-One Error

- Miscalculating the condition to end a loop.

  repeat while **i< = n or i<n**?

  repeat until **i=n ori>n**?

- Overlooking that an array of A[0] through A[n] contains n+1 elements.
- Cause:
  - Programmer is at fault.
  - Merging actual data with control data/metadata.

Incomplete Mediation

- **Mediation** means checking: the process of intervening to confirm an actor's authorization **before** it takes an intended action.
- Verifying that the subject is authorized to perform the operation on an object is called mediation.
- Incomplete mediation is a security problem that has been with us for decades.

Incomplete Mediation

1. http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=2015Jan17
2. http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=2015Feb30
3. http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=1500Jan17
4. http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=2095Abc47
5. http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212&parm2=SomeAbsurdValue123

Incomplete Mediation: Effects

- The system would fail catastrophically, with a routine's failing on a data type error.
- The receiving program would continue to execute but would generate a very wrong result.

Incomplete Mediation: Countermeasures

- Validate All Input. Try to anticipate them. Drop-down or choice boxes, etc.
- Do not leave sensitive data under control of an untrusted user.

```
http://www.things.com/order.asp?custID=101&part=555A&qy=20&price=10
&ship=boat&shipcost=5&total=205
```

```
http://www.things.com/order.asp?custID=101&part=555A&qy=20&price=1&
ship=boat&shipcost=5&total=25
```

- Solution is complete mediation.
- The three properties of a reference monitor are (1) small and simple enough to give confidence of correctness, (2) unbypassable, and (3) always invoked.
- These three properties combine to give us solid, complete mediation.

Time-of-check To Time-of-use Errors

- Involves synchronization.
- 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.
- Between access check and use, data must be protected against change.
- Example.
- It exploits the delay between the two actions: check and use.
- Between the time the access was checked and the time the result of the check was used, a change occurred, invalidating the result of the check.

Time-of-check To Time-of-use Errors

- **Security Implication**
    - Checking one action and performing another is an example of **ineffective access control**, leading to confidentiality failure and/or integrity failure.
- **Countermeasures**
    - Critical parameters are not exposed during any loss of control. The access-checking software must own the request data until the requested action is complete.
    - Ensure serial integrity, that is, to allow no interruption (loss of control) during the validation.
- All these protection methods are expansions on the tamperproof criterion for a reference monitor.

Malicious Code: Malware

- **Malicious code** or **rogue programs** or **malware**
- 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 or both.
- Malicious intent distinguishes from unintentional errors, even though both kinds can certainly have similar and serious negative effects.
- Malicious code can be directed at a specific user or class of users, or it can be for anyone.

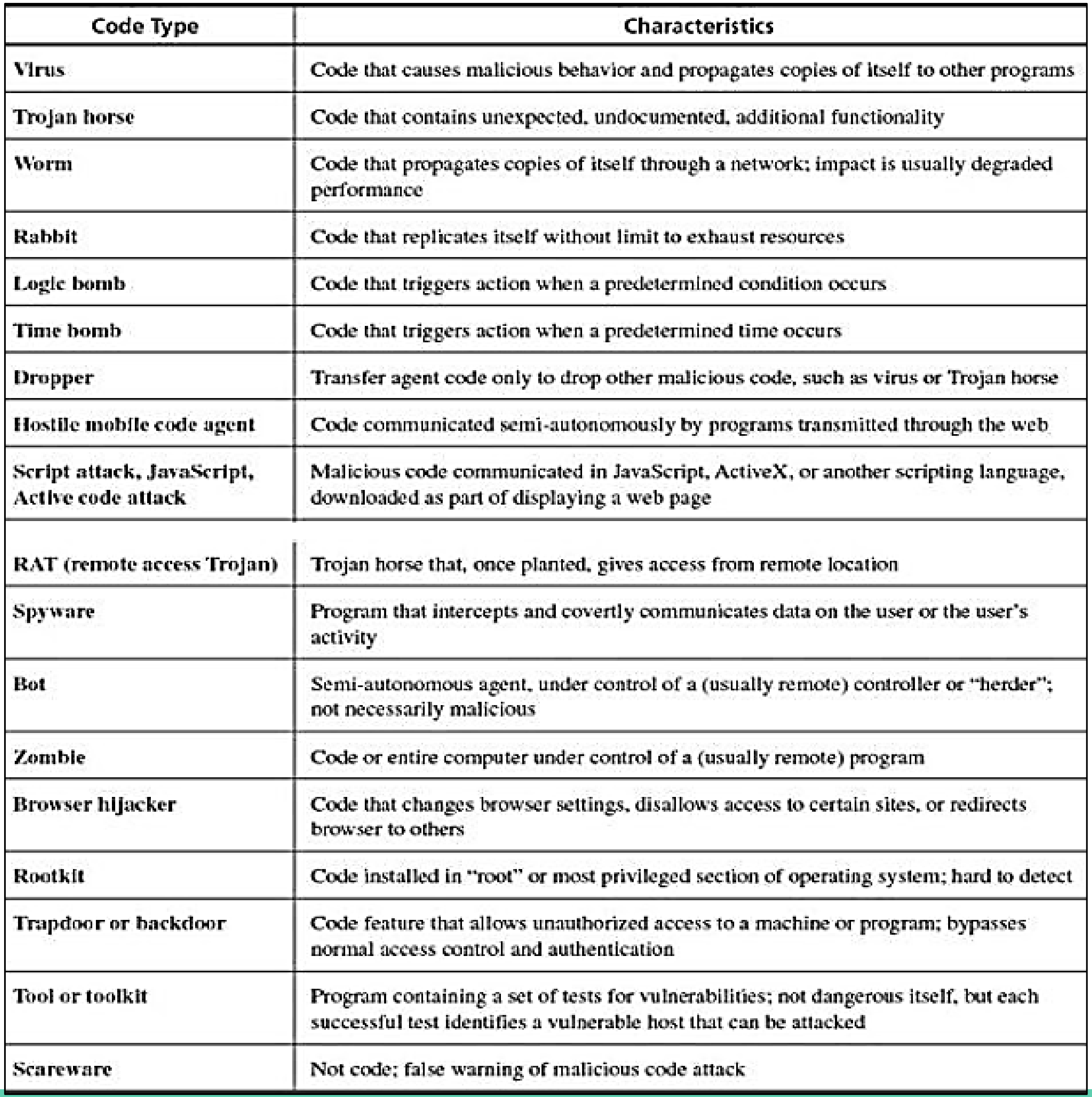
Why Worry About Malicious Code?

- Malicious code can do much harm.
  - Writing a message on a computer screen, stopping a running program, generating a sound or erasing a stored files.
- Malicious code has been around a long time.
  - Malicious code is still around and its effects are more pervasive.

Types of Malicious Codes

Code Type	Characteristics
Virus	Attaches itself to program and propagates copies of itself to other programs
Worm	Propagates copies of itself through a network
Trojan horse	Looks legal/normal programs, but contains unexpected, additional functionality
Logic bomb	Triggers action when condition occurs
Time bomb	Triggers action when specified time occurs
Trapdoor/backdoor	Allows unauthorized access to functionality
Rabbit	Replicates itself without limit to exhaust resources

Types of Malicious Code



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

Virus, Worm, and Trojan Horse

- A **virus** is a program that can replicate itself and pass on malicious code to other non-malicious programs by modifying them.
- It infects other healthy subjects by attaching itself to the program and either destroying the program or coexisting with it.
- The infection usually spreads at a geometric rate.
- Virus: Transient or Resident.
- A **transient** virus has a life span that depends on the life of its host.
- 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.

Virus, Worm, and Trojan Horse

- A **worm** is a program that spreads copies of itself through a network.
- What is the primary difference between a worm and a virus?
- A worm operates through networks, and a virus can spread through any medium.
- The worm spreads copies of itself as a stand-alone program, whereas the virus spreads copies of itself as a program that attaches to or embeds in other programs.
- Worms do have a common, useful purpose. Crawlers.

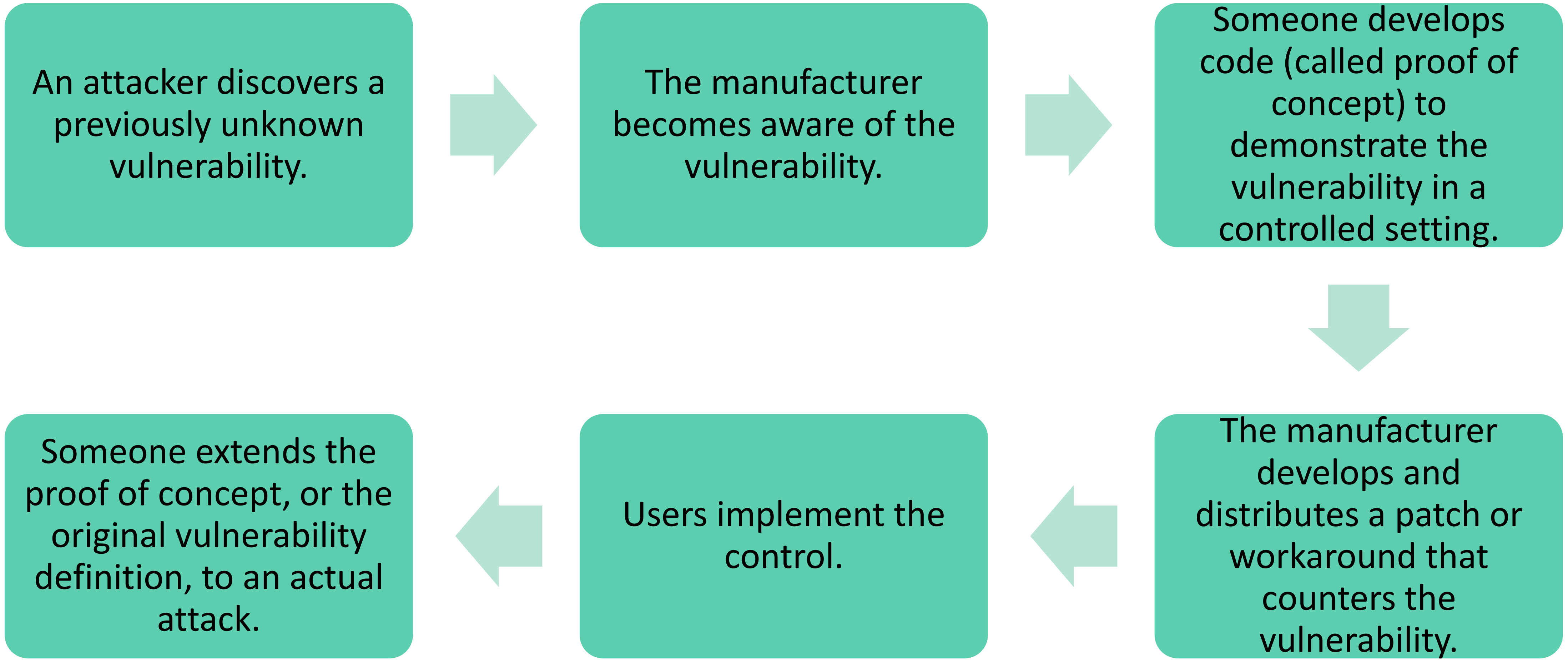
Virus, Worm, and Trojan Horse

- A **Trojan horse** is malicious code that, in addition to its primary effect, has a second, nonobvious, malicious effect.
- Trojan horse malware slips inside a program undetected and produces unwelcome effects later on.
- Example?
- Trojan horse is on the surface a useful program with extra, undocumented (malicious) features. It does not necessarily try to propagate.

Zero-day exploit: An attack before availability of the control.

Zero day attack: Active malware exploiting a product vulnerability for which the manufacturer has no countermeasure available.

General Exploit Timeline



An attacker discovers a previously unknown vulnerability.

The manufacturer becomes aware of the vulnerability.

Someone develops code (called proof of concept) to demonstrate the vulnerability in a controlled setting.

The manufacturer develops and distributes a patch or workaround that counters the vulnerability.

Users implement the control.

Someone extends the proof of concept, or the original vulnerability definition, to an actual attack.

Virus: Triggering

- For malware to do its malicious work and spread itself, it must be executed to be activated.
- A SETUP program that is run to load and install a new program on a computer.

  Virus code could be in the distribution medium. On execution, may install itself in a permanent storage or in any executing programs in memory.

  May or may not be human triggered.
- Email Attachment/ Attached File
- Document Virus
- Autorun

  Autorun is a feature of operating systems that causes the automatic execution of code based on name or placement.

Any other?

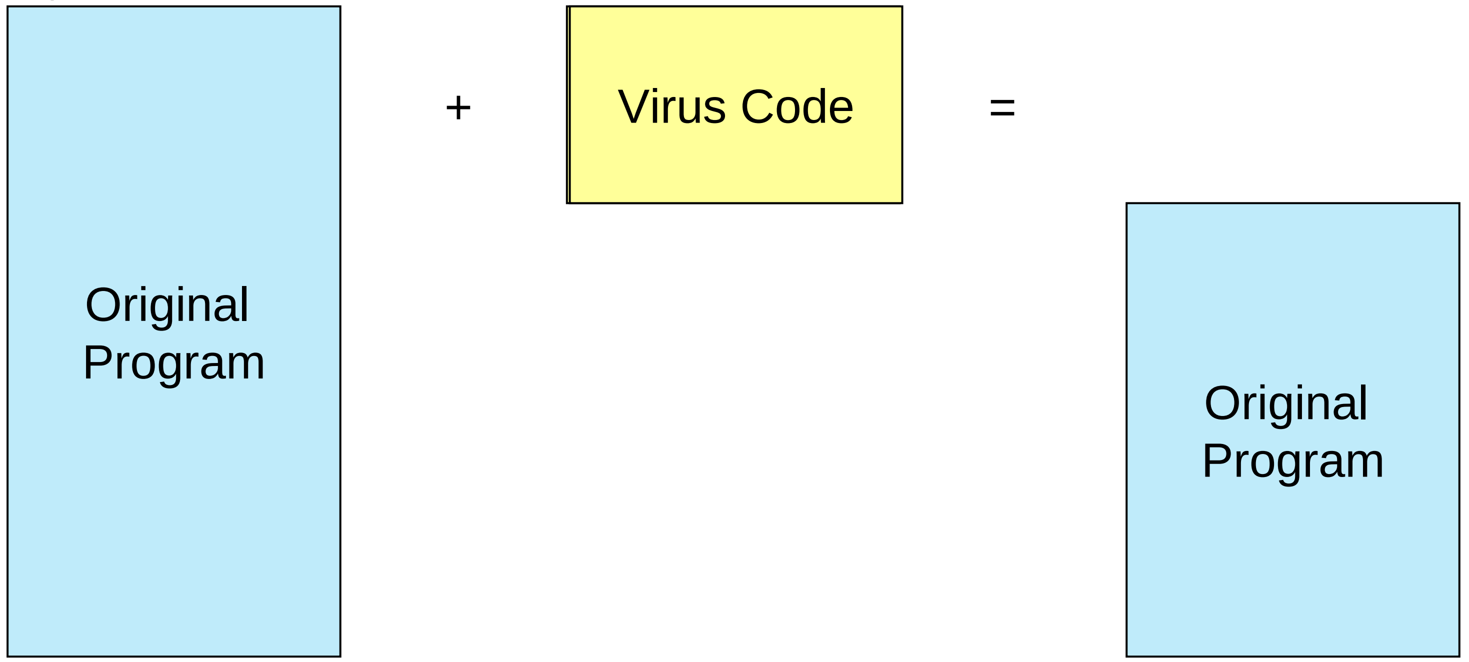
How Viruses Attach

- Appended viruses:
- Beginning
- Viruses that surround a program.
- Integrated viruses and replacement.

Appended Viruses

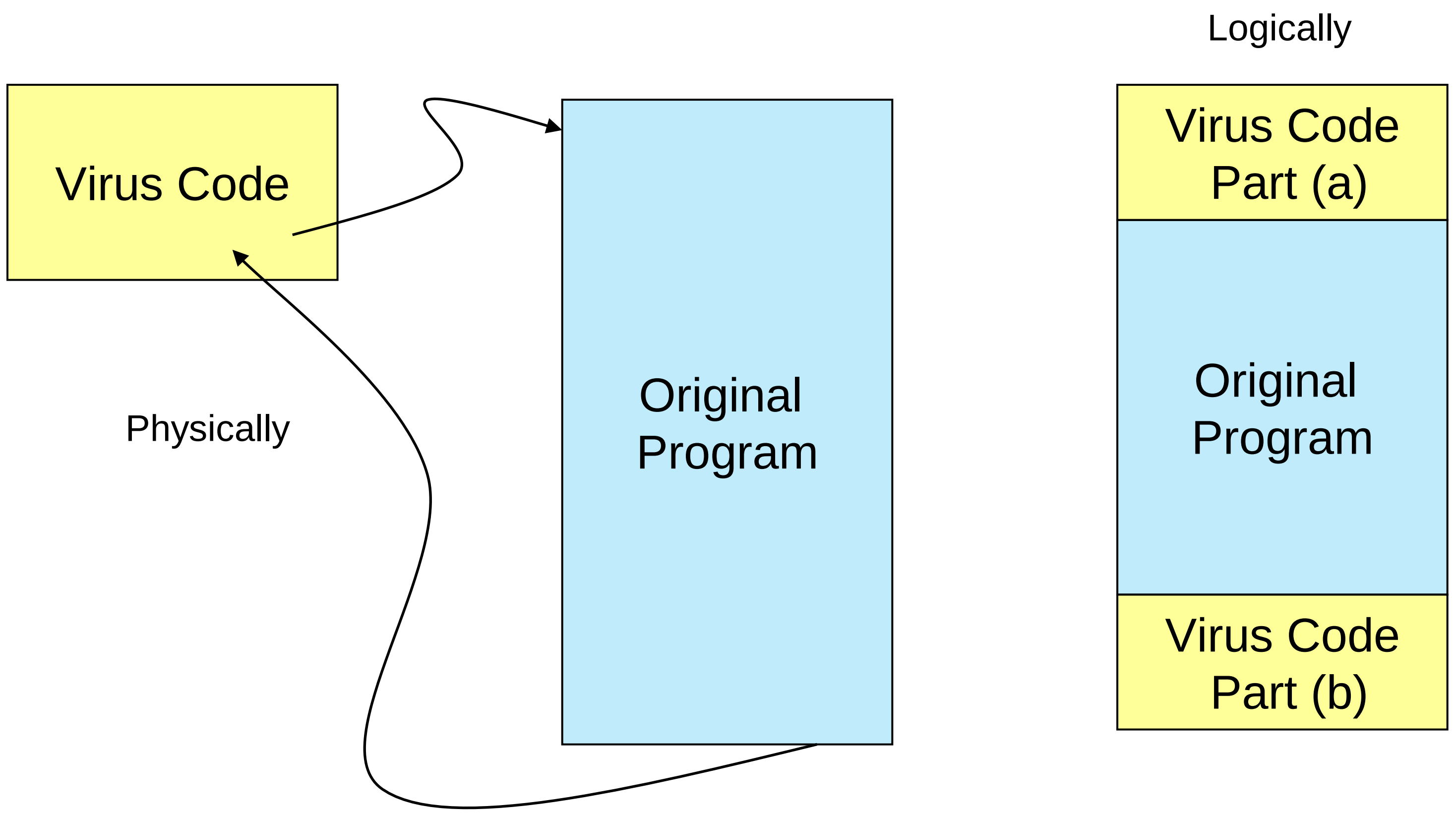
- A program virus attaches itself to a program. Then, whenever the program is run, the virus is activated.
- Usually easy to design and implement.

- Simple and usually effective.
- The virus writer need not know anything about the program to which the virus will attach.
- Often the attached program simply serves as a carrier for the virus.
- The virus performs its task and then transfers to the original program.



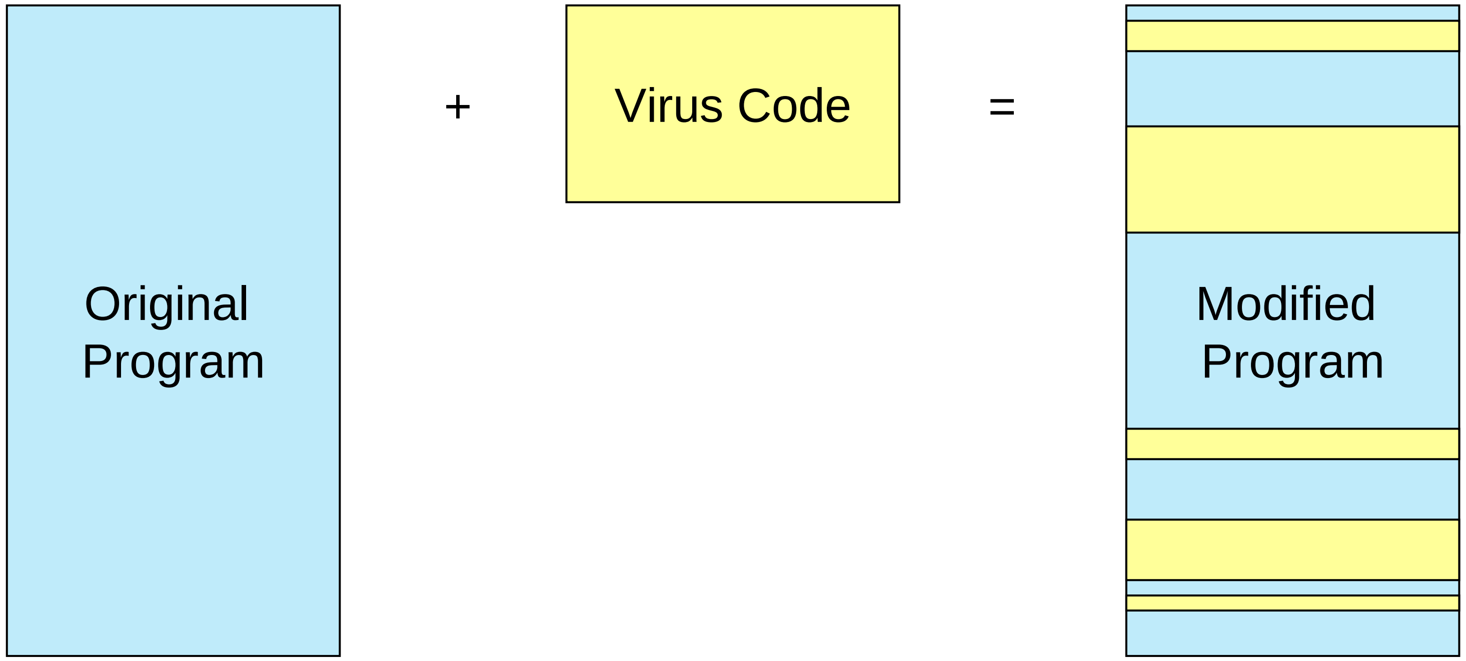
Viruses That Surround A Program

- Virus runs the original program but has control before and after its execution.



Integrated Viruses And Replacement

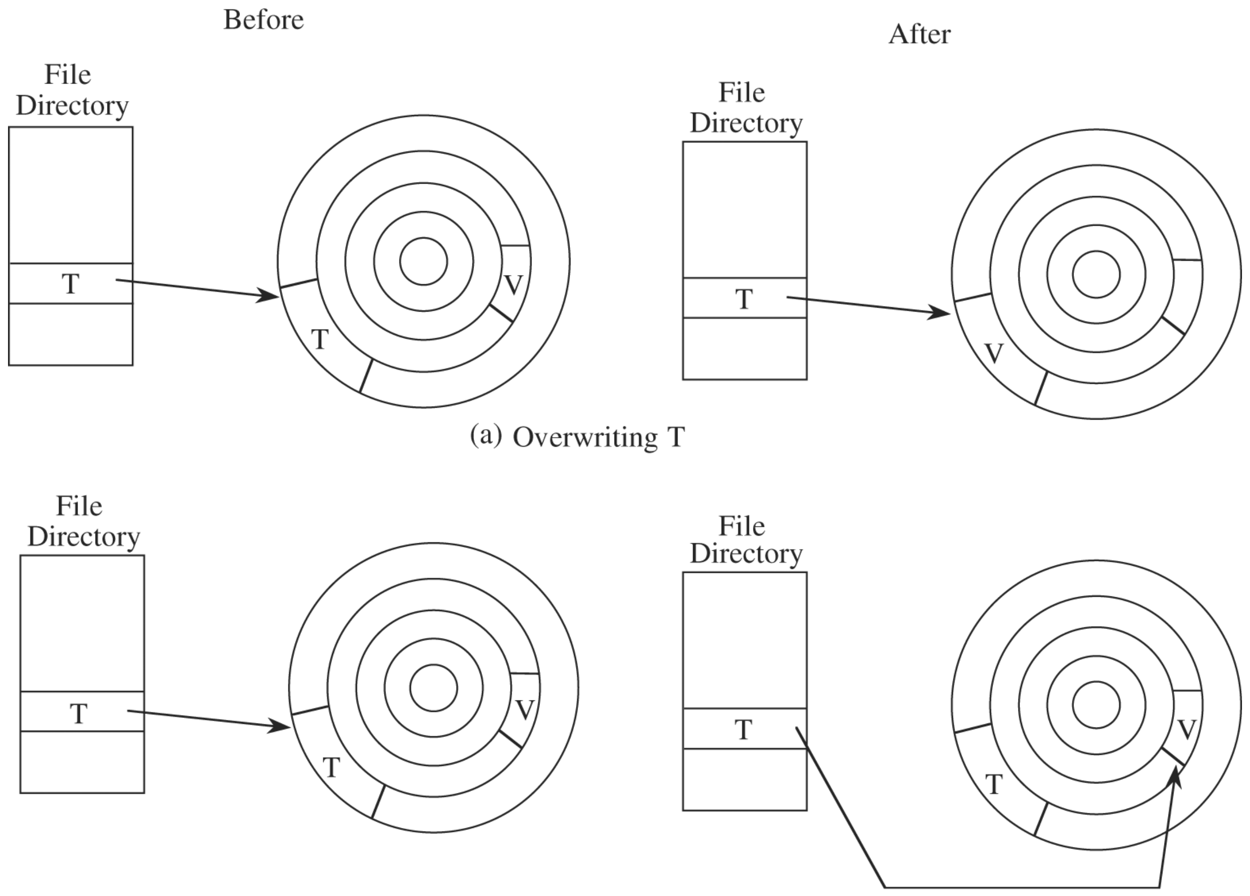
- Virus replaces some of its target, integrating itself into the original code of the target.
- The virus writer has to know the exact structure of the original program. Why?
- To know where to insert which pieces of the virus.



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).
- The virus has to either seem to be the target, or has to push the target out of the way and become a substitute itself.
- Invoked:
  - The virus can assume T’s name by replacing (or joining to) T’s code in a file structure.
  - The virus can overwrite T in storage.
  - 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.

How Malicious Code Gains Control



(a) Overwriting T

(b) Changing Pointers

Homes for Malware

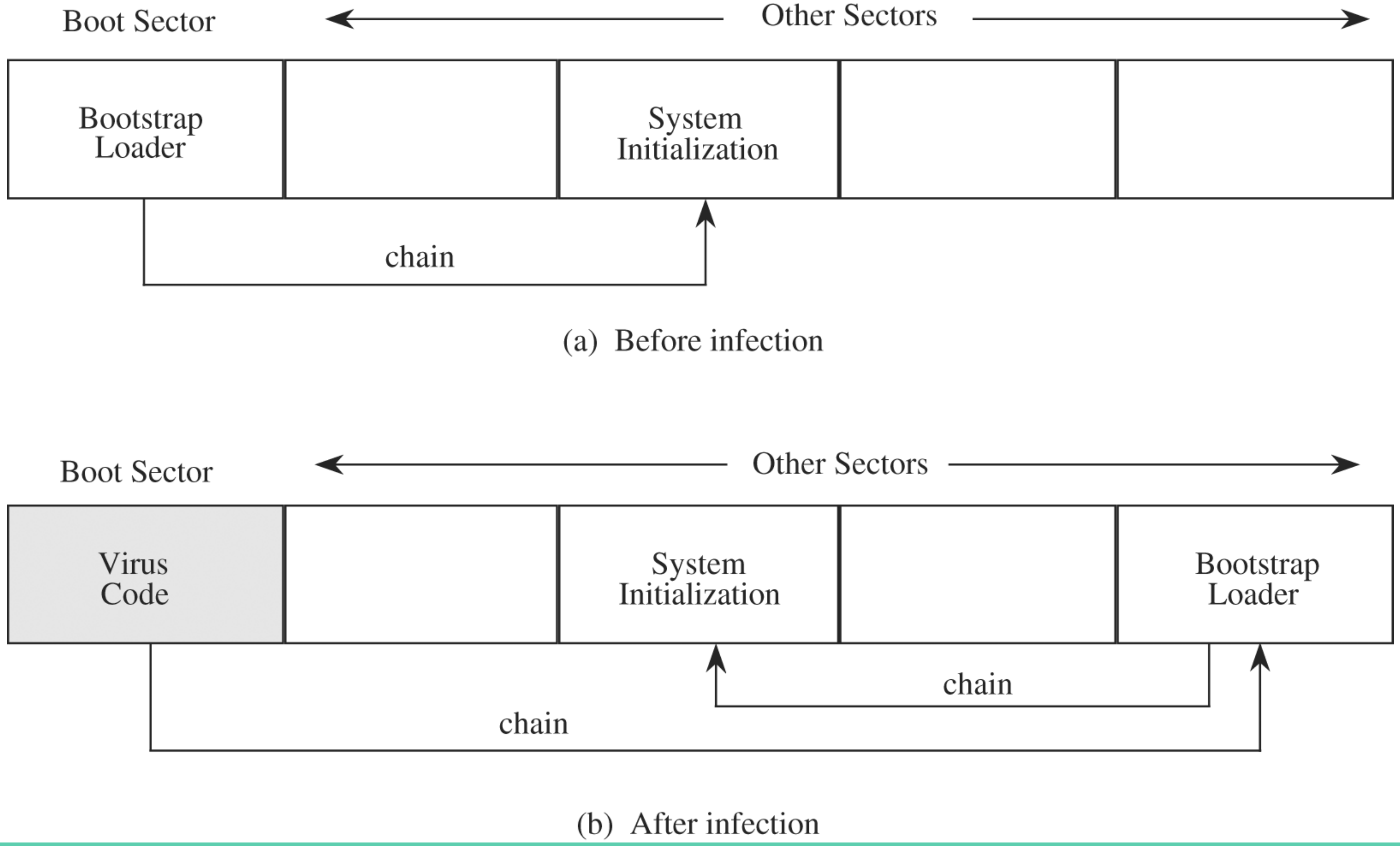
The virus writer may find these qualities appealing in a virus :

- It is hard to detect.
- It is not easily destroyed or deactivated.
- It spreads infection widely.
- It can re-infect its home program or the other programs.
- It is easy to create.
- It is machine independent and OS independent.

Homes for Malware

- **One-Time Execution (Implanting)**
    - Malicious code often executes a one-time process to transmit or receive and install the infection.
    - The first step is to acquire and install the code. It must be quick and not obvious to the user.
- **Boot Sector Viruses**
    - The virus gains control early in the boot process, before most detection tools are active, so that it can avoid, or at least complicate, **detection**.
    - OS, device handlers, and other necessary applications are numerous and have unintelligible names, so malicious code writers do not need to hide their code completely.

Boot Sector Viruses



Homes for Malware

- **Memory-Resident Viruses**
  - Frequently used parts of the OS and a few specialized user programs remains in memory and is called “resident” code.
  - Resident routines are sometimes called TSRs or “terminate and stay resident” routines.
  - Virus writers also like to attach viruses to resident code because the resident code is activated many times while the machine is running.
  - Each time the resident code runs, the virus gets activated too. It can then look for and infect uninfected carriers.
  - A virus can also modify the operating system’s table of programs to run. How is this vicious?

Other Homes for Malware

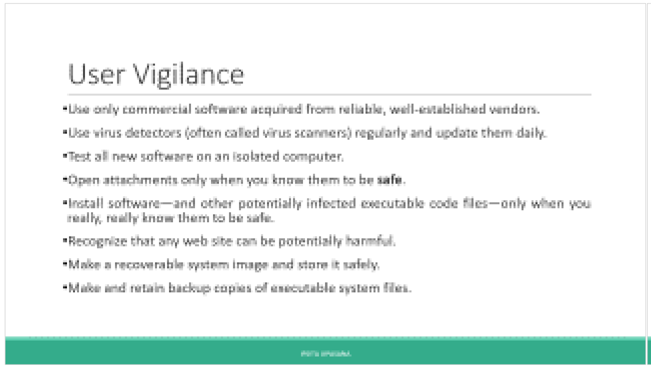
- Many applications, such as word processors and spreadsheets, have a “macro” feature, by which a user can record a series of commands and then repeat the entire series with one invocation.
- A virus writer can create a virus macro that adds itself to the startup directives for the application. It also then embeds a copy of itself in data files so that the infection spreads to anyone receiving one or more of those files.
- Code libraries
- Simple files like PDF (interpretive data) and their handlers (interpreter).
- If there is a flaw in the PDF interpreter or the semantics of the PDF interpretive language, opening a PDF file can cause the download and execution of malicious code.

Polymorphic Viruses

- A virus that can change its appearance is called a polymorphic virus. (Poly means "many" and morph means "form.")
- A two-form polymorphic virus can be handled easily as two independent viruses.
- Therefore, the virus writer **intent on preventing detection** of the virus will want either a **large or an unlimited number of forms** so that the number of possible forms is too large for a virus scanner to search for.
- A polymorphic virus has to **randomly reposition** all parts of itself **and randomly change all fixed data.**
- 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**.

Countermeasures for Users

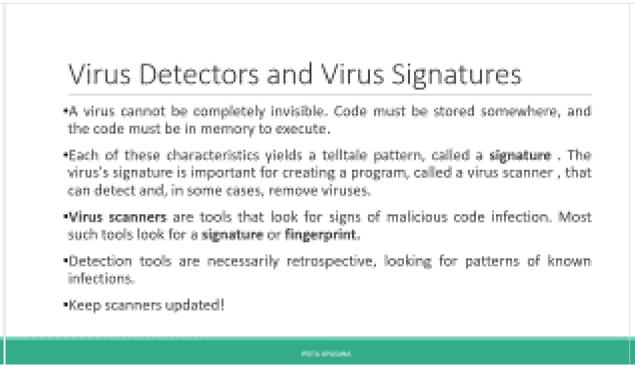
- User Vigilance
- Virus Detectors
- Virus Signatures
- Code Analysis



User Vigilance

- Use only commercial software acquired from reliable, well-established vendors.
- Use virus detectors (often called virus scanners) regularly and update them daily.
- Test all new software on an isolated computer.
- Open attachments only when you know them to be **safe**.
- Install software—and other potentially infected executable code files—only when you really, really know them to be safe.
- Recognize that any web site can be potentially harmful.
- Make a recoverable system image and store it safely.
- Make and retain backup copies of executable system files.

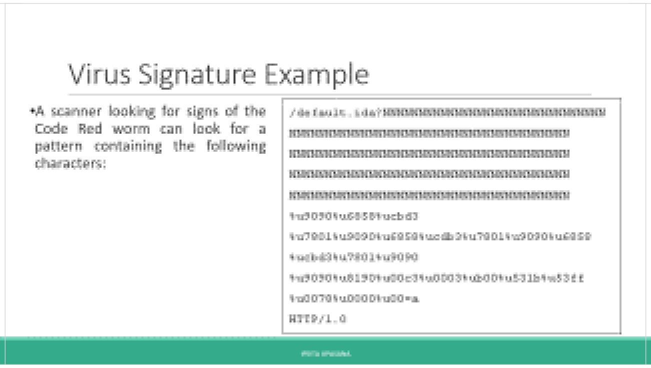
IPSITA UPASANA



Virus Detectors and Virus Signatures

- A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute.
- Each of these characteristics yields a telltale pattern, called a **signature** . The virus's signature is important for creating a program, called a virus scanner , that can detect and, in some cases, remove viruses.
- **Virus scanners** are tools that look for signs of malicious code infection. Most such tools look for a **signature** or **fingerprint.**
- Detection tools are necessarily retrospective, looking for patterns of known infections.
- Keep scanners updated!

IPSITA UPASANA

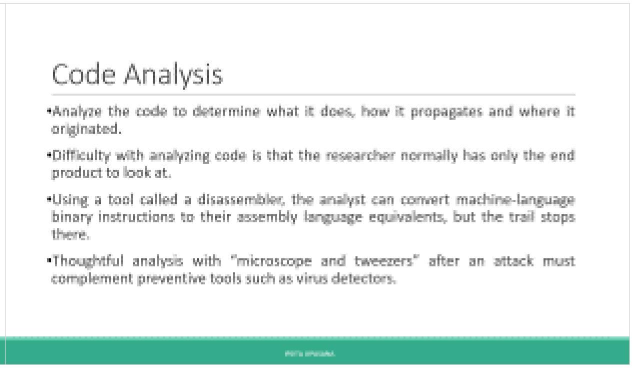


Virus Signature Example

- A scanner looking for signs of the Code Red worm can look for a pattern containing the following characters:

```
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3
%u7801%u9090%u6858%ucdb3%u7801%u9090%u6858
%ucbd3%u7801%u9090
%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff
%u0078%u0000%u00=a
HTTP/1.0
```

IPSITA UPASANA



Code Analysis

- Analyze the code to determine what it does, how it propagates and where it originated.
- Difficulty with analyzing code is that the researcher normally has only the end product to look at.
- Using a tool called a disassembler, the analyst can convert machine-language binary instructions to their assembly language equivalents, but the trail stops there.
- Thoughtful analysis with "microscope and tweezers" after an attack must complement preventive tools such as virus detectors.

IPSITA UPASANA

User Vigilance

- Use only commercial software acquired from reliable, well-established vendors.
- Use virus detectors (often called virus scanners) regularly and update them daily.
- Test all new software on an isolated computer.
- Open attachments only when you know them to be **safe**.
- Install software—and other potentially infected executable code files—only when you really, really know them to be safe.
- Recognize that any web site can be potentially harmful.
- Make a recoverable system image and store it safely.
- Make and retain backup copies of executable system files.

Virus Detectors and Virus Signatures

- A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute.
- Each of these characteristics yields a telltale pattern, called a **signature** . The virus's signature is important for creating a program, called a virus scanner , that can detect and, in some cases, remove viruses.
- **Virus scanners** are tools that look for signs of malicious code infection. Most such tools look for a **signature** or **fingerprint.**
- Detection tools are necessarily retrospective, looking for patterns of known infections.
- Keep scanners updated!

Virus Signature Example

- A scanner looking for signs of the Code Red worm can look for a pattern containing the following characters:

```
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3
%u7801%u9090%u6858%ucdb3%u7801%u9090%u6858
%ucbd3%u7801%u9090
%u9090%u8190%u00c3%u0003%ub00%u531b%u53ff
%u0078%u0000%u00=a
HTTP/1.0
```

Code Analysis

- Analyze the code to determine what it does, how it propagates and where it originated.
- Difficulty with analyzing code is that the researcher normally has only the end product to look at.
- Using a tool called a disassembler, the analyst can convert machine-language binary instructions to their assembly language equivalents, but the trail stops there.
- Thoughtful analysis with “microscope and tweezers” after an attack must complement preventive tools such as virus detectors.

Countermeasures for Developers

- Modularity, Encapsulation, and Information Hiding.
- Mutual Suspicion.
  - Mutually suspicious programs operate as if other routines in the system were malicious or incorrect.
  - A calling program cannot trust its called sub-procedures to be correct, and a called sub-procedure cannot trust its calling program to be correct.
  - Each protects its interface data so that the other has only limited access.

Countermeasures for Developers

- Confinement
  - Confinement is a technique used by an operating system on a suspected program to help ensure that possible damage does not spread to other parts of a system.
  - Since a virus spreads by means of transitivity and shared data, all the data and programs within a single compartment of a confined program can affect only the data and programs in the same compartment.
- Simplicity
- Testing
  - Reduce the likelihood or limit the impact of failures.

Case Studies

- The BRAIN Virus
- The Internet Worm
- Code Red

Book

- Pfleeger C. P., Pfleeger S. L. and Margulies J., Security in Computing (5e), Prentice Hall, 2015, Chapter 3.