Section [3.3] from Reference No. 1:

Viruses and Other Malicious Code

By themselves, programs are seldom security threats. The programs operate on data, taking action only when data and state changes trigger it. Much of the work done by a program is invisible to users who are not likely to be aware of any malicious activity. However, since users usually do not see computer data directly, malicious people can make programs serve as vehicles to access and change data and other programs.

Why Worry About Malicious Code?

None of us like the unexpected, especially in our programs. Malicious code behaves in unexpected ways, thanks to a malicious programmer's intention. We think of the malicious code as lurking inside our system: all or some of a program that we are running or even a nasty part of a separate program that somehow attaches itself to another (good) program.

Malicious code can do anything any other program can, such as writing a message on a computer screen, stopping a running program, generating a sound, or erasing a stored file. Or malicious code can do nothing at all right now; it can be planted to lie dormant, undetected, until some event triggers the code to act. The trigger can be a time or date, an interval (for example, after 30 minutes), an event (for example, when a particular program is executed), a condition (for example, when communication occurs on a network interface), a count (for example, the fifth time something happens), some combination of these, or a random situation. In fact, malicious code can do different things each time, or nothing most of the time with something dramatic on occasion.

Malicious code runs under the user's authority. Thus, malicious code can touch everything the user can touch, and in the same ways. Users typically have complete control over their own program code and data files; they can read, write, modify, append, and even delete them. And well they should. But malicious code can do the same, without the user's permission or even knowledge.

Kinds of Malicious Code

Malicious code or **rogue program** is the general name for unanticipated or undesired effects in programs or program parts, caused by an agent intent on damage. This definition excludes unintentional errors, although they can also have a serious negative effect. This definition also excludes coincidence, in which two benign programs combine for a negative effect. The **agent** is the writer of the program or the person who causes its distribution.

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 non malicious programs by modifying them. 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 all other connected systems.

A virus can be either transient or resident. A **transient virus** has a life that depends on the life of its host; the virus runs when its attached program executes and terminates when its attached program ends. (During its execution, the transient virus may spread its infection to other programs.) A **resident virus** locates itself in memory; then it can remain active or be activated as a stand-alone program, even after its attached program ends.

A **Trojan horse** is malicious code that, in addition to its primary effect, has a second, non - obvious malicious effect. As an example of a computer 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. In this example, the user sees only the login occurring as expected, so there is no evident reason to suspect that any other action took place.

A **logic bomb** is a class of malicious code that "detonates" or goes off when a specified condition occurs. A **time bomb** is a logic bomb whose trigger is a time or date.

A **trapdoor** or **backdoor** is a feature in a program by which someone can access the program other than by the obvious, direct call, perhaps with special privileges. For instance, an automated bank teller program might allow anyone entering the number 990099 on the keypad to process the log of everyone's transactions at that machine. In this example, the trapdoor could be intentional, for maintenance purposes, or it could be an illicit way for the implementer to wipe out any record of a crime.

A **worm** is a program that spreads copies of itself through a network. The primary difference between a worm and a virus is that a worm operates through networks, and a virus can spread through any medium (but usually uses copied program or data files). Additionally, 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. A **rabbit** is a virus or worm that self-replicates without bound, with the intention of exhausting some computing resource. A rabbit might create copies of itself and store them on disk in an effort to completely fill the disk, for example.

Code Type	Characteristics
Virus	Attaches itself to program and propagates copies of itself to other programs
Trojan horse	Contains unexpected, additional functionality
Logic bomb	Triggers action when condition occurs
Time bomb	Triggers action when specified time occurs
Trapdoor	Allows unauthorized access to functionality
Worm	Propagates copies of itself through a network

Code Type	Characteristics
Rabbit	Replicates itself without limit to exhaust resources

How Viruses Attach

For a virus to do its malicious work and spread itself, it must be activated by being executed. For example, recall the SETUP program that you initiate on your computer. It may call dozens or hundreds of other programs, some on the distribution medium, some already residing on the computer, some in memory. If any one of these programs contains a virus, the virus code could be activated(It is possible for execution to occur without human intervention, though, such as when execution is triggered by a date or the passage of a certain amount of time.) After that, no human intervention is needed; the virus can spread by itself.

A more common means of virus activation is as an attachment to an e-mail message. In this attack, the virus writer tries to convince the victim (the recipient of the e-mail message) to open the attachment. Once the viral attachment is opened, the activated virus can do its work. Other types of files are equally dangerous. For example, objects such as graphics or photo images can contain code to be executed by an editor, so they can be transmission agents for viruses.

Appended Viruses

A program virus attaches itself to a program; then, whenever the program is run, the virus is activated. 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. The virus performs its task and then transfers to the original program.

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.

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. Clearly, the virus writer has to know the exact structure of the original program to know where to insert which pieces of the virus. Finally, the virus can replace the entire target, either mimicking the effect of the target or ignoring the expected effect of the target

and performing only the virus effect. In this case, the user is most likely to perceive the loss of the original program.

Document Viruses

Currently, the most popular virus type is what we call the **document virus**, which is implemented within a formatted document, such as a written document, a database, a slide presentation, a picture, or a spreadsheet. These documents are highly structured files that contain both data (words or numbers) and commands (such as formulas, formatting controls, links). The commands are part of a rich programming language, including macros, variables and procedures, file accesses, and even system calls. The writer of a document virus uses any of the features of the programming language to perform malicious actions.

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 reinfect its home program or other programs.
- It is easy to create.
- It is machine independent and operating system independent.

One-Time Execution

The majority of viruses today execute only once, spreading their infection and causing their effect in that one execution. A virus often arrives as an e-mail attachment of a document virus. It is executed just by being opened.

Boot Sector Viruses

A special case of virus attachment, but formerly a fairly popular one, is the so-called **boot sector virus.** 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. The operating system is software stored on disk. Code copies the operating system from disk to memory and transfers control to it; this copying is called the bootstrap (often boot) load because the operating system figuratively pulls itself into memory by its bootstraps. The firmware does its control transfer by reading a fixed number of bytes from a fixed location on the disk (called the boot sector) to a fixed address in memory and then jumping to that address (which will turn out to contain the first instruction of the bootstrap loader). The bootstrap loader then reads into memory the rest of the operating system from disk. To allow for change, expansion, and uncertainty, hardware designers reserve a large amount of space for the bootstrap load. The boot sector on a PC is slightly less than 512 bytes, but since the loader will be larger than that, the hardware designers support "chaining," in which each block of the bootstrap is chained to (contains the disk location of) the next block. This chaining allows big bootstraps but also simplifies the installation of a virus. The virus writer simply breaks the chain at any point, inserts a pointer to the virus code to be executed, and reconnects the chain after the virus has been installed.

The boot sector is an especially appealing place to house a virus. The virus gains control very early in the boot process, before most detection tools are active, so that it can avoid, or at least complicate, detection. The files in the boot area are crucial parts of the operating system. Consequently, to keep users from accidentally modifying or deleting them with disastrous results, the operating system makes them "invisible" by not showing them as part of a normal listing of stored files, preventing their deletion. Thus, the virus code is not readily noticed by users.

Memory-Resident Viruses

Some parts of the operating system and most user programs execute, terminate, and disappear, with their space in memory being available for anything executed later. For very 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 was needed. Such code 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 does too. Once activated, the virus can look for and infect uninfected carriers. For example, after activation, a boot sector virus might attach itself to a piece of resident code. Then, each time the virus was activated it might check whether any removable disk in a disk drive was infected and, if not, infect it. In this way the virus could spread its infection to all removable disks used during the computing session. A virus can also modify the operating system's table of programs to run. On a Windows machine the registry is the table of all critical system information, including programs to run at startup. If the virus gains control once, 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 virus will return on the next system restart.

Other Homes for Viruses

One popular home for a virus is an application program. Many applications, such as word processors and spreadsheets, have a "macro" feature, by which a user can record a series of commands and repeat them with one invocation. Such programs also provide a "startup macro" that is executed every time the application is executed. 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.

Libraries are also excellent places for malicious code to reside. Because libraries are used by many programs, the code in them will have a broad effect.

Virus Signatures

A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute. Moreover, the virus executes in a particular way, using certain methods to spread. Each of these characteristics yields a telltale pattern, called a **signature**, that can be found by a program that looks for it. The virus's signature is important for creating a program,

called a virus scanner, that can detect and, in some cases, remove viruses. The scanner searches memory and long-term storage, monitoring execution and watching for the telltale signatures of viruses.

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 systems 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

Most virus writers seek to avoid detection for themselves and their creations. Because a disk's boot sector is not visible to normal operations (for example, the contents of the boot sector do not show on a directory listing), many virus writers hide their code there. A resident virus can monitor disk accesses and fake the result of a disk operation that would show the virus hidden in a boot sector by showing the data that should have been in the boot sector (which the virus has

moved elsewhere). There are no limits to the harm a virus can cause, the damage is bounded only by the creativity of the virus's author.

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. Thus, instead of containing the fixed (and therefore searchable) string "HA! INFECTED BY A VIRUS," a polymorphic virus has to change even that pattern sometimes. Just by moving pieces around, the virus writer can create enough different appearances to fool simple virus scanners. Once the scanner writers became aware of these kinds of tricks, however, they refined their signature definitions.

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. For these viruses, the decryption routine itself, or a call to a decryption library routine, must be in the clear so that becomes the signature.

Section [7.4] from Reference No. 1:

Firewall

A firewall is a device that filters all traffic between a protected or "inside" network and a less trustworthy or "outside" network. Usually a firewall runs on a dedicated device; because it is a single point through which traffic is channeled, performance is important, which means non firewall functions should not be done on the same machine. Because a firewall is executable code, an attacker could compromise that code and execute from the firewall's device. Thus, the fewer pieces of code on the device, the fewer tools the attacker would have by compromising the firewall. Firewall code usually runs on a proprietary or carefully minimized operating system.

The purpose of a firewall is to keep "bad" things outside a protected environment. To accomplish that, firewalls implement a security policy that is specifically designed to address what bad things might happen. Part of the challenge of protecting a network with a firewall is determining which security policy meets the needs of the installation.

Types of Firewalls

Firewalls have a wide range of capabilities. Types of firewalls include

- packet filtering gateways or screening routers
- stateful inspection firewalls
- application proxies
- guards
- personal firewalls

Each type does different things; no one is necessarily "right" and the others "wrong

Packet Filtering	Stateful Inspection	Application Proxy	Guard	Personal Firewall
Simplest	More complex	Even more complex	Most complex	Similar to packet filtering firewall
Sees only addresses and service protocol type	Can see either addresses or data	Sees full data portion of packet	Sees full text of communication	Can see full data portion of packet
Auditing difficult	Auditing possible	Can audit activity	Can audit activity	Canand usually doesaudit activity
Screens based on connection rules	Screens based on information across packetsin either header or data field	Screens based on behavior of proxies	Screens based on interpretation of message content	Typically, screens based on information in a single packet, using header or data
Complex addressing rules can make configuration tricky	Usually preconfigured to detect certain attack signatures	Simple proxies can substitute for complex addressing rules	Complex guard functionality can limit assurance	Usually starts in "deny all inbound" mode, to which user adds trusted addresses as they appear

What Firewalls Can and Cannot Block

Firewalls are not complete solutions to all computer security problems. A firewall protects only the perimeter of its environment against attacks from outsiders who want to execute code or access data on the machines in the protected environment.

- Firewalls can protect an environment only if the firewalls control the entire perimeter. That is, firewalls are effective only if no unmediated connections breach the perimeter. If even one inside host connects to an outside address, by a modem for example, the entire inside net is vulnerable through the modem and its host.
- Firewalls do not protect data outside the perimeter; data that have properly passed (outbound) through the firewall are just as exposed as if there were no firewall.
- Firewalls are the most visible part of an installation to the outside, so they are the most attractive target for attack. For this reason, several different layers of protection, called **defense in depth**, are better than relying on the strength of just a single firewall.

- Firewalls must be correctly configured, that configuration must be updated as the internal and external environment changes, and firewall activity reports must be reviewed periodically for evidence of attempted or successful intrusion.
- Firewalls are targets for penetrators. While a firewall is designed to withstand attack, it is not impenetrable. Designers intentionally keep a firewall small and simple so that even if a penetrator breaks it, the firewall does not have further tools, such as compilers, linkers, loaders, and the like, to continue an attack.
- Firewalls exercise only minor control over the content admitted to the inside, meaning that inaccurate data or malicious code must be controlled by other means inside the perimeter.

Firewalls are important tools in protecting an environment connected to a network. However, the environment must be viewed as a whole, all possible exposures must be considered, and the firewall must fit into a larger, comprehensive security strategy. Firewalls alone cannot secure an environment.

Reference:

Charles P. Pfleeger, Shari Lawrence Pfleeger, **Security in Computing, 4th Edition, Pearson.**