Chapter 1

Introduction

To be deemed *trustworthy*, a computer system should

- exhibit all of the functionality users expect,
- not exhibit any unexpected functionality, and
- be accompanied by some compelling basis to believe that to be so,

despite failures of system components, attacks, operator errors, and the inevitable design and implementation flaws found in software. Thus, *computer security*, which focuses on resisting attacks and is the subject of this text, concerns just one of the factors that undermine computer system trustworthiness. It is a fascinating intellectual discipline, a source of challenging engineering problems, and an area of increasing practical importance.

Security breaches are growing in frequency, sophistication, and consequence.[1] This growth is likely driven by our increasing dependence on computing systems—as individuals and as nations—which makes these systems attractive targets for malfeasants who design and launch attacks to bring about intended adverse results. Analogous incentives do not exist for the other factors that undermine trustworthiness: system component failures are caused by natural events (e.g., earthquakes, thunderstorms, and errant alpha particles striking computer chips) whose rates are beyond mortal control; operator errors are correlated with training, competence, and user-interface; and design and implementation errors are related to the size and complexity of system relative to developer expertise.

Beyond Byzantium. Some might argue that implementing *Byzantine fault-tolerance* should suffice for resisting attacks because, by definition, it involves tolerating hardware and software faults whose manifestations range from benign to arbitrary and malicious behavior. This reasoning oversimplifies:

[1] There is no authoritative data on successful attacks and their impact. Some victims are unable to reveal data about attacks, since it undermines public trust; others have incentives to overstate the costs and consequences. Anecdotal accounts, however, suggest that alarm about the trends is not misplaced.

- Byzantine fault-tolerance methods almost exclusively focus on ensuring that correct system outputs are produced in a timely manner. Other important security properties, such as confidentiality and providing evidence to convince third parties about the provenance of system outputs, are typically ignored.

- Byzantine fault-tolerance methods invariably employ some form of replication. Replication is antithetical to confidentiality, because having more copies brings attackers more opportunities for compromise. Furthermore, even when confidentiaility of the replicated state is not a concern, cryptographic keys might still have to be stored and kept confidential at replicas. For example, such keys are needed if irrefutable evidence about the provenance of outputs is desired.

- Byzantine fault-tolerance methods assume the failure of one replica is unlikely to effect another—a reasonable assumption because physically-separated components connected only by narrow bandwidth channels do tend to fail independently. However, an attack that compromises a single replica, if repeated at others, would likely compromise them as well. We conclude that when done in a straightforward way, replication improves fault-tolerance but does not enhance a system's tolerance to attack.

Byzantine fault-tolerance methods must be extended for building systems that resist attacks. Computer security methods provide the basis for these extensions, giving ways to build individual components that resist attack and giving ways to reduce the chances a successful attack on one component could also be successful at others.

1.1 Attacks, Threats, and Vulnerabilities

Attacks against a system are mounted by motivated capable adversaries, known as *threats* or *attackers*, who attempt to violate the system's security properties by exploiting *vulnerabilities*—unintended aspects of a system's design, implementation, or configuration. That is a rather succinct description, so we now explore each of these elements in more detail.

Threats. Avoid the temptation to design defenses first and only afterwards characterize the threats they resist, because it risks producing a system that defends against the wrong adversary. The first (not last) step in building a secure system should be to decide on the threats.

A list of threats, based on work by a U. S. Defense Science Board task force, appears as Figure 1.1. Threats there are ordered from least to most pernicious. Better resourced threats (in terms of funding, talent, time, or organizational support) are generally considered the more dangerous, but attacker motivation or passion can substitute for funding. Note that the cost of designing an attack could differ substantially from the cost of launching that attack. Expertise

- Incomplete, inquisitive, and unintentional blunders.
- Hackers driven by technical challenges.
- Disgruntled employees or customers seeking revenge.
- Criminals interested in personal financial gain, stealing services, or industrial espionage.
- Organized crime with the intent of hiding something or financial gain.
- Organized terrorist groups attempting to influence U.S. policy by isolated attacks.
- Foreign espionage agents seeking to exploit information for economic, political, or military purposes.
- Tactical countermeasures intended to disrupt specific weapons or command structures.
- Multifaceted tactical information warfare applied in a broad orchestrated manner to disrupt a major military mission.
- Large oganized groups or nation-states intent on overthrowing a government.

Figure 1.1: Taxonomy of Cybersecurity Threats

and/or access to secret information required to devise an attack might be expensive, but once an attack is created, it can be packaged and distributed (perhaps using the web) in ways that allow virtually anyone to launch it.

Not included in the taxonomy of Figure 1.1 are insiders who unwittingly assist in attacking a system. The term *social engineering* is used for attacks that employ human interaction and trickery to cause some outcome an attacker seeks. An adversary, for example, might exploit people's natural willingness to take action that helps solve what seems to be a pressing problem: The attacker poses as a new technical manager, telephones the victim, relates a fictitious story about urgent system problems, and requests the insider's password so an account can be "reset"; the insider divulges the password to be helpful, and the attacker now has a way onto the system. One defense against social engineering is educating the workforce about acceptable and unacceptable behavior, but many find it stifling to work in an environment where co-workers are not permitted to depart from standard operating procedures. So there is a tension between defending against social engineering and fostering the flexibility that is so important in running an enterprise, be it an army or a business.

How might potential threats for a given system be identified? There is, unfortunately, no easy answer. Knowledge about services a system renders or data

it stores can provide some insight about threats, though. Who profits from using or abusing the service or data? Also, a new computing system probably has some (perhaps manual) predecessor that it is replacing or augmenting; threats to that predecessor are likely also to be threats for the new computing system. Another rule of thumb is that higher-resourced threats are attracted by systems associated with high-value assets. Be mindful, though, to incorporate the social, political, and economic climate when estimating the value of an asset. For example, new fears about domestic terrorism might prompt a government to re-evaluate and withdraw from public view documents that, though helpful to the general population, could facilitate terrorist attacks.

Sources of Vulnerabilities. An obvious source of vulnerabilities is errors in a system's design or implementation. Eliminating all of these is a worthy goal but likely an unachievable one. This is because identifying software errors is costly in developer time, computer time, and delay to market. The cost of finding and removing that next bug becomes prohibitive at some point.

Vulnerabilities also result from implicit assumptions made by developers, and this is the basis for many well known attacks. One example are *buffer-overflow* attacks. Here, a developer assumes that some input will always fit into a specific-sized region of memory. In C programs, copying a value (e.g., a string) bigger than the destination buffer accommodates will overflow into adjacent memory, thereby changing other parts of the program state. One form of buffer-overflow attack causes a C routine to copy an attacker-provided input that overflows into a return address; the subsequent return operation loads the program counter with that attacker-provided address, which hijacks control and transfers to whatever routine the attacker had specified. Note, the assumption being made by the developer about input size is not only invalid but is entirely unnecessary—a C program can (and should) check the size of an input before copying it.

Assumptions made by developers sometimes are necessary, are explicit, and need not be limited to coding details. Algorithms often depend on assumptions about timing, failure manifestations, message delivery order, and other properties of the execution environment or system services. These assumptions must hold for the algorithm—and any systems that use it—to work correctly, so any means of violating such assumptions constitutes an attack. For example, a *denial of service* attack could be used to increase system load, causing critical tasks to be delayed so assumptions about timing are violated.

All else equal, systems whose correct operation is predicated on weaker sets of assumptions exhibit fewer vulnerabilities. With fewer vulnerabilities, the system is more resistant to attack. This defense does have a price, though. Algorithms that depend on weaker sets of assumptions are often more expensive and typically more complicated. Furthermore, some functionality simply cannot even be implemented unless certain assumptions are guaranteed to hold. So there is a limit on adopting weak-assumptions as a defense.

System configuration is another noteworthy source of vulnerabilities. Mod-

ern software systems are quite flexible, employing configuration files to customize each installation. These files are created not by the system's developers but by local site administrators who are less familiar with the system and must work from (often cryptic) documentation. Site administrators sometimes get it wrong and expose functionality more broadly than intended, effectively disabling defenses and giving users accesses that should be blocked; and sometimes site administrators just admit defeat and deploy whatever default configuration file accompanied the software distribution, even though historically such configuration files have allowed virtually unrestricted access to functionality.[2] Notice, unlike the vulnerabilities discussed above, system configuration vulnerabilities are not under the control of software developers and, in fact, can differ from one installation to the next.

As should now be clear, any non-trivial system is going to have vulnerabilities of one sort or another. Fortunately, repairing or even finding all vulnerabilities is not necessary for a software system to be considered secure. Some vulnerabilities might not be exploitable because attacks do not exist; exploiting others might be beyond the capabilities of your threats. Focus only on vulnerabilities that could be exploited by your threats. Each such vulnerability must be found and removed or else means should be deployed to limit damage possible from attacks that exploit the vulnerability. But note that believing a vulnerability to be unexploitable and ignoring it is risky—new attacks are developed all the time, and vulnerabilities that are unexploitable today could become exploitable tomorrow.

1.2 Security Properties

Systems for different tasks are typically expected to satisfy different *security policies*, which prescribe what must be done and what must not be done. A system for storing top-secret documents, for example, must prevent adversaries from learning the contents of those documents, whereas a system for managing bank accounts should ensure account balances change only in response to specific events: customer deposits and withdrawals, debits for bank fees, credits for interest payments, and so on.

Security policies legislate behavior by people, computers, executing programs, communications channels, and other system entities capable of taking action. Having a single term to denote any such entity is convenient; the term *principal* is conventionally used for this in the security literature. A principal acts on its own or it *speaks for* (equivalently *acts on behalf of*) another principal. For instance, a computer acts on behalf of the program it is executing; a keyboard speaks for the person who is typing on it; a communications channel speaks for (i) the computer that outputs messages onto that channel, (ii) the program that sends messages to that channel, as well as (iii) the user who caused that program to be executed.

[2]Software producers are starting to become senstive to this problem, and increasingly sensible defaults are being distributed with software.

Security policies themselves are typically formulated in terms of the three basic kinds of *security properties*:

> **Confidentiality (or secrecy).** Which principals are allowed to learn what information. []
>
> **Integrity.** What changes to the system (stored information and resource usage) and to its environment (outputs) are allowed. []
>
> **Availability.** When must inputs be read or outputs produced. []

These classes are not completely independent. For example, enforcing the integrity property that output y be calculated from input x could conflict with enforcing a confidentiality property stipulating that reading y reveals nothing about the value of x. As a second example, observe that any confidentiality property can be satisfied given a weak enough availability property because a system that does nothing has no way for attackers to learn information. Clearly, care must be taken in writing security policies to ensure the result is neither contradictory nor trivial.

1.2.1 Confidentiality

Among the more familiar confidentiality properties are those enforced by restricting which principal may read data that is stored in a file or region of memory. Sometimes even the existence of the data might need to be kept secret. Past patients at an alcoholism rehabilitation facility, for instance, might want to keep that history confidential, which implies keeping secret the very existence of their treatment files. So one typically finds that not only will an operating system restrict which ordinary files each principal can read but it will also restrict which directories each principal can read.[3]

Reading an object is only one way to learn information about that object. Inference is another. Through *information flow*, a principal might learn the value of one variable by reading another. The program fragment below, where variables *pub* and *priv* each store a single bit, illustrates.

$$pub := 0; \;\; \textbf{if } priv = 1 \textbf{ then } pub := 1$$

There are no assignments from *priv* to *pub* yet reading *pub* after execution of this fragment reveals the value of *priv*, so information flows from *priv* to *pub*.

Another way to learn information is by measuring some aspect of system behavior, called a *covert channel*, known to be correlated with information an attacker seeks. For example, a program might intensively access a system disk only after reading a certain value from a confidential variable and not after reading other values; an attacker's program concurrently accessing that disk

[3] A directory contains metadata (including the file name) for a set of files. The directory itself must be read in order to access any file in that set.

could then infer something about the value of that confidential variable by attempting I/O to the disk and observing the delay.

Finally, information can be learned by making inferences from statistical calculations. Suppose an attacker seeks to learn the value of an attribute that some database stores about an individual. A query to compute an average or other statistic on that attribute over all individuals seems as though it should preserve confidentiality of information associated with any single individual. Yet, this need not be the case. For example, a query to compute the average salary over a sub-population of size 1 yields the salary of the sole individual in that sub-population. And creating that sub-population of size 1 is surprisingly easy—gender, date of birth, and zip code (well known attributes for an individual) together uniquely identify 99% of the people in Cambridge, Massachusetts. Moreover, even when we cannot create a sub-population of size 1, by re-submitting a query to different overlapping projections of the database, an attacker can extract an attribute value corresponding to the unique individual in their intersection by analyzing results returned for the various sub-populations.

Where Privacy Fits. We distinguish between confidentiality and privacy.[4] Confidentiality is a security property; privacy is a right.

> **Privacy.** The right of an individual to determine what personal information is communicated to which others, and when. []

A key insight for understanding privacy is to appreciate that whether a piece of information is considered personal by the subject often depends on context—who is learning the information and their need to know it. For example, many would regard their clothes size to be personal information; they would consider it a privacy violation if this information becomes known to colleagues but have no hesitation about revealing it to sales clerks in clothing stores. People also tend to be more concerned about privacy if they believe the entity receiving their personal information is not trustworthy, because they then have reason to fear abuse or loss of control over the subsequent spread of that information.

For computing systems, privacy often is concerned with so-called *personally identifiable information* (PII), which encompasses information that potentially can be used to identify, contact, or locate a person. Examples of PII include a person's name, social security number, telephone number, address, and so on.

Keeping information confidential is not the only way to respect a person's privacy. Personal information can be disclosed without violating a subject's privacy (i) by giving that subject notice of the possible disclosure when that information is first collected or (ii) by obtaining the subject's consent prior to revealing that information. Thus, supporting the right of privacy requires more than mechanisms for implementing confidentiality properties. It requires a means to notify subjects and to obtain their consent; both of which likely will involve machinery outside the scope of a computing system.

[4] Some regard confidentiality and privacy as synonyms. Our purposes in this text are better served by having distinct meanings for these two terms.

1.2.2 Integrity

Integrity properties proscribe specified "bad things" from occurring during execution, where a "bad thing" is something that, at least in theory, could be observed—a finite sequence of instructions, a state, or a history of states.[5] Integrity properties thus include many of the usual notions of program correctness, such as correctly computing outputs from inputs, absence of program-exceptions during execution, and mutual exclusion of critical sections. Limits placed on the use of real or virtual resources during execution also are integrity properties, since exceeding those limits is a "bad thing" that can be attributed to some finite execution that required the resource.

Integrity properties can be used to convey proscriptions about data and how it is changed, enabling a data-centric view of security. So, an integrity property could specify that changes to a data item be made by running a specific routine, that updates preserve a global consistency constraint, or that specified checks be made before allowing an update. To enforce such properties, operating systems typically provide control over write and execute access to files and memory regions.

Write and/or execute access are not sufficient for enforcing all integrity properties, though. We might want to specify, for example, that high-integrity data not be contaminated by low-integrity data—a restriction concerning information flow and not (necessarily) access control. This kind of integrity property is useful in defending against corruption that might result when content from different sources is combined. This integrity property is also useful for defending against certain classes of attack: to defend against malicious code downloaded from the Internet, we label as low integrity anything obtained from the Internet, and we label as high integrity all local content; to defend against buffer-overflow attacks, we label as low integrity any inputs provided by a user, and label as high integrity the program counter, return address locations, function pointers, and other containers for code addresses. We then prevent attackers from gaining control of the system by enforcing an integrity policy that prevents low integrity content from affecting high integrity resources.

1.2.3 Availability

Availability properties prescribe that a "good thing" happens during execution. For this definition, one essential characteristic of a "good thing" is that it need not have finite duration; it therefore could be any finite or infinite sequences of instructions or states. The other essential characteristic of the "good thing" in an availability property is that of being required, in contrast to a safety property's "bad thing" which is being prohibited. Availability properties include

[5] Confidentiality properties are decidedly different from integrity properties, but an integrity property might imply a confidentiality property. An example is the integrity property stipulating that some principal not be permitted to perform a read operation on a file F. The "bad thing" is execution of the read. If executing a read is the only way to learn the contents of F then this integrity property implies the confidentiality property that the contents of F be kept secret.

aspects of program correctness, such as execution terminates (useful for a system call), execution does not terminate (useful for an operating system), and requests are processed in a fair manner (useful for a server). The latter two examples illustrate a "good thing" that cannot be associated with an identifiable point in execution. They also illustrate that violation of an availability property sometimes defines a safety property.

Once the poor cousin to confidentiality and integrity, availability properties are growing in importance as networked computing systems have become widespread:

- With the advent of the web, business is increasingly conducted over networks. Availability properties are what ensure a business can communicate with its customers and partners.

- Critical infrastructures, like electric power and gas distribution, air and rail transportation, as well as banking and financial markets, all are increasingly being monitored and controlled using networked systems. Availability properties enable the control functions to work.

- The military is starting to embrace *network-centric warfare*, a doctrine involving networked systems to link commanders and troops with surveillance and weapons platforms. Availability properties enable situational awareness and facilitate the timely deployment of both defensive and offensive assets.

In all of these applications, compromises to availability have significant financial or life-threatening, if not tactical and strategic, consequences. Availability can no longer receive short shrift when building a secure system.

1.3 Assurance Matters

An *assurance argument* provides evidence that a system will behave as intended. Ideally, that evidence will be compelling. But the work required to completely analyze systems of even moderate size can be prohibitive, so we often must settle for assurance arguments that guarantee weaker properties of system behavior, that concern only a portion of the system, or that merely increase our confidence that the system will behave as intended rather than guaranteeing it.

We might base an assurance argument on the system itself, the process used to create the system, or the personnel who participated in that process. Arguments based on process or personnel are typically less work to construct but are less convincing. That your developers have passed a certification examination or that the development process employed was like one successfully used before involves little, if any, knowledge about the system of concern. And an argument that largely ignores how or why a system works cannot compel belief that the system's behavior will be as intended. So we should prefer assurance arguments that depend on system details.

A broad spectrum of approaches to assurance arguments have been developed, ranging from formal methods to software analyzers to testing. The approaches differ in what kinds of properties they address and how convincing is the evidence they provide. They also differ in the amount, distribution, and kinds of effort each involves; some require significant initial human effort before yielding evidence to support any increase in confidence, while others yield useful results from the very start. One thing they all share, however, is that the cost of increased assurance grows with system size, and establishing high levels of assurance for systems of even moderate size is today far beyond our capabilities.

Finally, it is worth recalling that we desire systems that cannot be subverted by specified threats, which is not quite the same as resisting specified attacks. Nobody knows how to derive all attacks a threat might initiate, and it is unlikely such deductions could ever be automated. Thus, while an assurance argument might provide evidence that the system will defend against certain attacks, there remains a gap between what our analyses reveal and the statement about threats we seek.

What is Trusted?

Whether a component behaves as intended is determined, in part, by whether components on which it depends behave as intended—guarantees about the one component involve assumptions about others. We appropriate terms *trust* and *trusted* in order to make explicit such assumptions, saying if a component C depends on C' then C *trusts* C' or equivalently C' is *trusted by* C. For example, we might say that a word processor trusts the file system, or we might be more precise and say exactly what file system functionality is trusted by the word processor. Should the file system be trusted only to preserve the integrity of files it stores, then we imply the word processor's confidentiality properties are not compromised if the file system reveals file contents to attackers (perhaps because the word processor encrypts files before storing them on the file system).

When a component is trusted, two kinds of assumptions are introduced. The first is that operations provided by that component's interfaces behave as expected. The second are a set of concomitant assumptions about the component's internals, including (i) any advertised confidentiality and integrity properties on the state and (ii) proscriptions about whether and when the component invokes operations elsewhere in the system. That is, stipulate that some component is trusted and you are stipulating that certain attacks on the component will not succeed and that other attacks (anywhere in the system) can have only limited impact on the operation of this component.

Being trusted is not the same as being trustworthy. Trustworthy components by definition function as intended, whereas trusted components need not behave as assumed—trust can be misplaced. Since attackers rightly see assumptions as being potential vulnerabilities, misplaced trust creates vulnerabilities. In particular, if a component C trusts C', then a vulnerability in C' is potentially also a vulnerability for C, and success in attacking C' might well be a way to compromise C. So we should prefer components that are trustworthy to those

that are (merely) trusted.

It is easier to have confidence in the correct operation of artifacts we can understand, and it is easier to understand artifacts that are smaller and simpler. This leads to the following often stated but too often ignored principle for building trustworthy components.

> **Principle: Economy of Mechanism.** Prefer mechanisms that are simpler and smaller, hence easier to understand, easier to get right, and easier to have confidence that they are right. []

Economy of Mechanism implies that a mechanism involving fewer control paths should be preferred, because humans are capable of enumerating and analyzing only small numbers of execution trajectories. It also implies that general-purpose mechanisms should be preferred to collections of special-purpose mechanisms, again due to the mental energy required to understand one mechanism versus many.

Trusted Computing Base. In any non-trivial system, some aspect of the system's behavior is going to be considered more critical. It is often a small set of security properties but can be virtually any property. The set of mechanisms (along with any associated configuration files) required to support that critical functionality is known as the *trusted computing base* (TCB).

We should endeavor to ensure that the TCB is trustworthy. The thinking behind Economy of Mechanism applies here: Keep the TCB simple and small, so that (i) it will be easier to understand and (ii) the cost of constructing an assurance argument will not be prohibitive. This also suggests a sensible yardstick for comparing alternative system designs—the design having the simpler and smaller TCB should be preferred.

1.4 Enforcement Principles

Successfully attacking a computer causes the target to execute instructions that it shouldn't, resulting in violation of some security property. The instructions might come from a program provided by the attacker. Or they might be code already at the target being invoked with unexpected inputs or in an unexpected state.

An *enforcement mechanism* must either prevent that execution or recover from its effects. To succeed at this, attackers must be unable to (i) replace or modify the code that implements the enforcement mechanism, (ii) circumvent the enforcement mechanism, or (iii) alter files or data structures used by the enforcement mechanism. Finally, although any enforcement mechanism will necessarily support only a limited space of policies efficiently, we should strive for *separation of policy and mechanism* and prefer mechanisms where changing from one policy to another within that space is easily accommodated.

To make this a bit more concrete, consider how today's operating systems enforce confidentiality and integrity properties on files.

- Associated with each file is an *access control list*, which enumerates those principals allowed to read the file and those principals allowed to write that file.
- Operating system routines are the sole way to perform file I/O. A file I/O request is rejected by the operating system unless the principal issuing that I/O request appears on the appropriate access control list.
- Replacement, modification, and circumvention of the file I/O routines is prevented by mechanisms that protect the operating system's integrity; attacker changes to access control lists is prevented by storing these lists in files and defining suitable access control lists for those.

Notice the extent to which separation of policy and mechanism is exhibited by this enforcement mechanism. In order to change which principals can access a file, it suffices to change an access control list but code need not be changed. However, some policies that might be of interest cannot be enforced with this mechanism. These include policies where access authorization is determined by past accesses (e.g., no principal can change a file after some principal has read that file).

1.4.1 Defending Against Attack

The basic strategies available for protecting against attacks are few in number and rather straightforward. We survey them in general terms here, giving details in the later chapters that discuss specific enforcement mechanisms.

Isolation. This strategy admits a range of implementations. The extreme case is physically isolating the system by locating it (including all its input/output devices) inside a large metal vault, with a power feed being the only electrical connection to the outside world.[6] The vault's metal walls form a Faraday cage, which prevents transmission of signals (including electronic noise produced by the computer's circuitry, which might reveal information about a computation in progress) from traveling into or out of the vault. Terminals and printers located inside the vault are thus the only way to communicate with programs running on the computer, and attackers are blocked from physically entering the vault.

Less extreme forms of isolation are more typical and often more useful. Rather than sequestering the computer within a vault, software is used to create isolation by restricting communication between programs, subsystems, or systems. The term "communication" should be interpreted here rather broadly to mean the ability of one principal to influence execution by another. Restrict the ability of attackers to communicate with their targets, and we block attacks

[6] Prior to the advent of wireless networking, computer security experts would speak of an "air gap" as being the ultimate protection mechanism. A vault is the limit case, and this technology is still used in highly sensitive national security applications. Such a facility in the U.S. is sometimes known as a SCIF (Sensitive Compartmented Information Facility).

because, by definition, an attack influences execution of its target (by causing instructions to be executed).

We see software-implemented isolation, for example, in the following operating system abstractions:

> **Virtual Machines.** A *virtual machine* behaves as if it were an isolated computer despite other execution on the underlying hardware. A *hypervisor* (or *virtual machine manager*) implements virtual machines that have the same instruction set as the underlying hardware. With *paravirtualization*, the hardware's non-privileged instructions appear unaltered, so applications needn't be modified for execution on the virtual machine, but privileged instructions and the memory architecture may differ, so systems software might have to be modified. Some virtual machines implement instruction sets bearing no resemblance to what the underlying hardware provides. This facilitates program portability across different hardware platforms; software to support the given virtual machine is written for each platform. For example, Sun's Java programming language is defined in terms of the Java Virtual Machine (JVM), which was designed with the slogan "Write once, run anywhere" for Java programs in mind. []
>
> **Sandboxes.** For software executed inside a *sandbox*, all operations on the environment's resources are redirected to shadow copies of those resources. This shields the real instances of resources from the effects of attacks perpetrated by the sandboxed software (but also from the effects of any other execution, thereby limiting the utility of the sandboxed software). Sandboxing is easily implemented when instructions manipulating the environment's resources are among those that cause traps. Web browsers and email clients often implement sandboxing to protect the system they run in against attacks conveyed in web pages and attachments. []
>
> **Processes.** System software, known as a *kernel* (also called a *supervisor* or *nucleus*), is employed to multiplex a real processor and create a set of *processes*. Each process executes in its own isolated address space; kernel-supported non-privileged instructions provide access to system services and a set of shared resources. The shared resources reduce isolation by providing direct and indirect means for one process to affect execution by others. This is by design—the process abstraction is intended as a building block for implementing larger systems, and process coordination (the antithesis of isolation) is often necessary for an ensemble to achieve system-wide goals. []

Virtual machines, then, are protected from being harmed by their environment, sandboxes prevent harm to the envionment, and processes implement some of both forms of isolation.

Weaker forms of isolation are better suited to how most of us use computers today. We run programs, like web browsers and email clients, whose sole function is to communicate with principals not running locally. Other of the

applications we execute are designed to communicate with each other through a shared file system, so a task can be accomplished by dividing it into smaller subtasks that each can be handled using a single application. And our graphical user interfaces provide operations like cut-and-paste, to facilitate transfer of information from one program's window to another, and provide operations like double-clicking, to invoke programs based on selected text in another program's window. None of these regimes is consistent with strong isolation. Indeed, computing as we know it today might have to change quite radically to embrace strong isolation.

All we really require is just enough isolation to block communication used for attacks. That suggests deploying mechanisms to filter channels of communication. Unfortunately, incomplete solutions are the best we can hope for here. If a communication channel can convey a program, then our filter must determine whether that program implements some known attack, which requires that the filter decide the equivalence of two[7] programs, an undecidable problem. Another source of incompleteness arises because attacks will continue to be developed after the filter has been deployed; attacks that had not been anticipated might not be detected and blocked.

Incomplete solutions for detecting and blocking communications that convey attacks are nevertheless widely used in practice. Here are two well known examples.

> **Firewalls.** A *firewall* interrupts the connection from an enclave of computers to some network. The firewall is configured to pass only certain messages, typically blocking those destined to ports associated with applications that should not be accessed from outside the enclave. For instance, we might locate the corporate web server outside the enclave comprising that company's desktop computers and then configure the firewall to block outside requests for web content (i.e., requests to port 80) from reaching the desktop computers. The desktop computers are then no longer subject to attacks that target the port expected to run a web server. []

> **Code Signing.** With *code signing*, provenance—that is, who produced the content—becomes a criteria for deciding whether that content is safe to execute. System software only loads for execution content with a preapproved provenance. Cryptographic digital signatures are used to protect the integrity of the content and to identify its producer, hence the name "code signing". Microsoft's Authenticode™ for example protects Internet Explorer from web pages containing malicious executable content by allowing content downloaded from the web to be executed only if it was produced by Microsoft or by a Microsoft-approved software producer. []

Viewed abstractly, isolation plays the same role in computer security as did the tall, imposing perimeter walls in protecting a medieval city from marauders.

[7] One program is the known attack and the second program is the one found in the communication.

Add openings that are too large or too numerous, and those walls cease to be an effective defense. Yet, having those openings facilitates activities—commerce and other interactions with outsiders—that we might want to encourage. Note the tension between defending the city and promoting the daily activities of its citizens.

An analogous tension exists when isolation is used for computer security. A firewall, for example, is less effective for isolation when it is configured to pass more different kinds of messages to more different ports, yet there is usually considerable pressure to do just this. For instance, B2B (business-to-business) e-commerce creates such pressure because computers on different sides of firewalls must now have direct access to each other's applications and data. As a second example, code signing becomes less valuable as larger numbers of software producers are approved, because chances are then increased that an approved producer will distribute code containing vulnerabilities. Yet, there will be pressure—from code producers who want to sell products and consumers who want to use them—for the list of approved producers to be large.

Isolation is thus best suited to situations where (i) there is little pressure to puncture the boundaries that isolation defines and (ii) communication that does cross those boundaries is limited and carefully prescribed. As we shall see, protecting the integrity of enforcement mechanisms often turns out to be exactly this kind of a situation, as does enforcing certain kinds of confidentiality properties.

Monitoring. Because attacks (by definition) involve execution, a second means of defense can be to monitor a set of interfaces and halt execution before any damage is done using operations those interfaces provide. Three elements comprise this defense:

- a security policy, which prescribes acceptable sequences of operations from some set of interfaces;
- a *reference monitor*, which is a program that is guaranteed to receive control whenever any operation named in the policy is requested, and
- a means by which the reference monitor can block further execution that does not comply with the policy.

By prescribing what is acceptable, the security policy implicitly defines executions that are not acceptable; these are the attacks this defense addresses. Note that monitoring can be used to implement isolation in settings where operations are the sole way principals communicate—the monitor serves as the filter on the communications channels. Thus, implementations for the isolation schemes discussed above often embody forms of monitoring.

The characterization of monitoring given above leaves much about implementation unspecified. Various approaches can be employed. Policies concerning the interface between the processor and memory can be enforced by installing a reference monitor as part of the kernel trap-handler for memory-access faults;

this allows policies that restrict memory accesses (read, write, or execute) to be enforced. Policies involving shared hardware resources (such as the interval timer) and shared OS abstractions (such as the file system) can be enforced by installing a reference monitor in the kernel and having it run whenever a process invokes the corresponding OS operation. Later chapters explore in greater detail these and various other schemes.

Which interfaces we choose to monitor depends on what attacks we wish to defend against. A safe, but perhaps extreme, solution is:

> **Principle: Complete Mediation.** The reference monitor intercepts every access to every object. []

Even if Complete Mediation is not always practical, it is generally a good starting point. Analysis of the system and threats can then provide a justification for not monitoring specific objects or operations. Were we concerned, for instance, with attacks that violate confidentiality properties by sending information out the network, then the obvious interfaces to monitor include those for sending messages and any other interfaces that initiate network traffic, such as interfaces for reading and writing from network file servers, interfaces to send email, interfaces to print on network-accessible printers, and if the paging device is located across the network, then even interfaces whose operations can cause pages to be evicted from memory.

Systems commonly use monitoring to support policies formulated in terms of principals and *privileges*. The security policy specifies: (i) an assignment[8] of privileges to principals, and (ii) an enumeration of what privilege(s) a principal must possess for each specific operation. A policy $\mathcal{P}$ is considered *stronger* than another policy $\mathcal{Q}$ if $\mathcal{P}$ assigns some principals fewer privileges than $\mathcal{Q}$ does and/or additional privileges are required by $\mathcal{P}$ for some operations to occur than are required by $\mathcal{Q}$. Thus, stronger policies rule out more behaviors.

By ruling out possible execution, stronger policies protect against more attacks. What we should seek, then, is the strongest policy that still enables the system to accomplish its goals. That ideal policy is characterized as follows.

> **Principle: Least Privilege.** A principal should be only accorded the minimum privileges it needs to accomplish its task. []

As an illustration, consider the design of a spell checker module to augment a text editor. A correctly operating spell checker likely needs read access to the file being accessed by the text editor, read access to the dictionary of correct spellings, but requires access to no other files; a correctly operating text editor needs read and write access to the user's files, but it does not itself need access to the dictionary.

The damage an attack causes by subverting a program depends on what privileges that program has, just as the damage a user causes with carelessly entered commands depends on what privileges that user has. The more privileges a principal has, the more damage that principal can inflict, so ignoring the

[8] This assignment of privileges might be dynamic, changing as execution proceeds.

Principle of Least Privilege is risky in a world where programs have vulnerabilities and people make mistakes. As a concrete example, the UNIX super-user is a principal that has read, write, and execute access to all files. In most versions of UNIX, certain system programs have super-user privileges (helpful, for example, to the system program that delivered mail by storing it in each user's directory); also, operators and UNIX system programmers log-on as super-user to manage the system. This means that an attacker who subverted the mail-delivery program could write (hence, delete) any file in the system, and a systems programmer who entered a command to delete all files in the current directory might crash the system if that command was accidentally typed while in a working directory that stored the system's executables.

The Principle of Least Privilege is impossible to implement if the same privilege suffices for multiple different objects or operations. We should endeavor to avoid that.

> **Principle: Separation of Privilege.** Different accesses should require different privileges. []

However, putting Separation of Privilege into practice can be a nightmare. In a computer system with separate privileges for every object and every operation, somebody will have to decide who should be given the millions of privileges, and every principal will have to manage the privileges it receives. Few of us would have the patience to allocate or acquire all of those privileges manually, and there has been remarkably little progress in creating automated support or suitable user interfaces to help.

We might be tempted to address these practical difficulties by exploring alternative representations for a principal's privileges. If virtually every principal is being granted a specific privilege, then why not just list the prohibited principals rather than listing the (much larger number of) principals being granted the privilege? In theory, both representations should be equivalent, since the one could be computed from the other. In practice, however, these two representations are significantly different, because people make mistakes and it is people who currently decide what privileges are given to each principal. Consider the two possible mistakes—(i) mistakenly prohibiting access by a principal versus (ii) mistakenly granting it. By mistakenly prohibiting access, some task that should work might not, which will lead to a complaint (and presumably redress); by mistakenly granting access, operations that should be blocked will run, which is unlikely to be detected and could violate a system security property. The second kind of mistake seems far worse than the first, and that suggests the following.

> **Principle: Failsafe Defaults.** The presence of privileges rather than the absence of prohibitions should be the basis for determining whether an access is allowed to proceed. []

Recovery. Attacks whose effects are reversible could be allowed to run their course, if a recovery mechanism were available afterwards to undo any dam-

age. This defense is quite different from those based on blocking execution or blocking communication—recovery embodies an optimistic outlook that allows all execution to proceed, whereas blocking takes the more conservative stance of prohibiting any potentially harmful execution. The two different approaches are compatible, though, and they might well be employed in concert.

The effects of only some attacks can be reversed, so recovery is not always a feasible defense. We gain some insight into where recovery is useful by considering attacks whose effects can and cannot be reversed.

Confidentiality Violations. A secret that has been disclosed is no longer confidential. If that secret is a statement about the world, then its disclosure to an adversary is unlikely to be reversible. Troop strength, the formula for Coca-Cola, or an individual's medical records are examples of such secrets.

Disclosure of some secrets, such as passwords and cryptographic keys, can be remediated by choosing replacements. By selecting a new login password immediately after the old one becomes known to an adversary, attackers are limited to a short window during which they can access the system and cause damage. But replacing an encryption key that becomes known is less effective—although messages encrypted under the new key cannot be decrypted by attackers, messages that were encrypted under the old key, intercepted by the attacker, and saved, can still be read by the adversary. []

Integrity Violations. Changes to internal system state are usually reversible, so recovery can be used to defend against attacks whose sole effect is to change that state. Special system support is typically required to perform such recovery. Transactions[9] are an ideal packaging for state changes that might have to be reversed, but transactions are not well suited for structuring all applications and can have an unacceptable impact on performance.

An alternative to transactions is simply to take frequent backups of the system state. This imposes virtually no restrictions on application structure, but creating and storing frequent backups can have a non-trivial impact on performance. Also, the backups must be available and not subject to corruption by attackers. We might prevent such corruption by storing backups off-line and by not using any software on the compromised system when restoring state from a backup (so contamination from the prior attack is not perpetuated even if the attacker had managed to modify the compiler, loader, or other system software to produce tainted outputs despite having uncorrupted inputs). Backups are particularly ef-

[9] Recall, by definition, a transaction might abort and, therefore, the run-time must support an undo operation to reverse the transaction's state changes. To enable recovery from state changes caused by attacks, such undo functionality would have to be extended so that previously committed transactions could also be aborted.

fective for defending against attacks that install software for facilitating subsequent attacker access.

Attacks that produce outputs affecting the physical environment can be hard to reverse. Erroneously issuing a check, launching a missile, or re-routing an airplane to a new destination are examples of outputs that cannot be reversed, although one can imagine compensating actions for each: a stop-payment could be issued on the check, the missile could be ordered to self-destruct, or the airplane's course could be re-adjusted back to the old destination. Compensating actions don't always exist, though, and might be prohibatively expense when they do. []

Availability Violations. For a system not involved in sensing or controlling the physical environment, recovery from availability violations could be feasible: evict the attacker and resume normal processing.[10] Note that buffered inputs will queue until normal processing resumes, which means higher than usual loads until the backlog has been processed. []

Defense in Depth. No single mechanism is likely to resist all attacks. So the prudent course is that system security depend on a collection of complementary mechanisms rather than trusting a single mechanism. By *complementary*, we mean that mechanisms in the collection

- exhibit *independence*, so any attack that compromises one mechanism would be unlikely to compromise the others, and
- *overlap*, so that attackers can succeed only by compromising multiple mechanisms in the collection.

Both of these requirements are easier to state than to satisfy, because they quantify over all attacks, including attacks not yet known. Even so, a carefully considered defense in depth is, in practice, apt to be stronger than using a single mechanism in isolation, if the above requirements are approximated with sufficient fidelity.

One example of defense in depth is seen when you withdraw cash at an automated teller machine (ATM) that checks for both a valid bank card (a token presumably held only by the rightful card holder) and a PIN (a 4 digit Personal Identification Number presumably known only to the rightful card holder). The bank considers it unlikely that somebody who steals your bank card will deduce your PIN (which depends on your having selected a non-obvious PIN and not writing that PIN on the card itself), so the two different checks probably satisfy the independence requirement. The overlap requirement is addressed by the bank requiring both checks be satisfied before allowing the cash withdrawal.

As another example, we might employ both a firewall and a sandbox to defend against attacks conveyed in email attachments. The firewall modifies

[10] In a system for controlling a reactor or an airplane, delaying the delivery of the outputs could lead to a catastrophic failure, because physics won't wait.

packets it handles, deleting email attachments having types that, when opened, execute; the sandbox blocks executing attachments from reading files, writing files, and invoking certain programs (e.g., to initiate communications over the Internet). Arguably, the two mechanisms satisfy the independence requirement—one mechanism modifies packets while the other blocks executions; the overlap requirement is satisfied because each attachment passes first through the firewall and only then is executed in the sandbox.

Independence of the constituent mechanisms is the hardest part of implementing defense in depth. Empirical evidence suggests that diverse mechanisms are less likely to share vulnerabilities but, lacking a concrete definition of diversity, that observation is less useful than it might at first seem. In theory, two mechanisms having any point of similarity cannot be considered diverse, because an attack that exploits vulnerabilities present in a point of similarity could compromise both mechanisms. This is illustrated by bank card example above, where both mechanisms have the card holder in common and there is a trivial attack that subverts both mechanisms: abduct the card holder and use coercion to get the bank card and learn the PIN. In practice, mechanisms deployed in the same system will necessarily have points of similarity, although not all these similarities will have exploitable vulnerabilities. So, lacking a scientific basis for deciding which similarities could be exploitable, experience and judgement must be the guide for implementing defense in depth.

1.4.2 Secrecy of Design

There are good reasons to keep information about defenses secret, and there are good reasons not to. Consequently, the utility of what is variously known as *secrecy of design* or *security by obscurity*[11] has been a topic of considerable debate.

Pro: Proponents argue that withholding details about design or implementation makes attacking a system that much more difficult. Anything that makes the attacker's job harder constitutes a useful defense, so secrecy of design adds one more layer to a defense in depth.

Con: Opponents point out that attackers will learn design and implementation secrets sooner or later. Making system details public increases the chances that system vulnerabilities will be identified, so they can be repaired.

Each position involves some implicit assumptions. By exposing these assumption, we can better understand circumstances where secrecy of design makes sense.

The first of the implicit assumptions concerns the feasibility of actually keeping design and implementation details secret. In some environments, secrets don't stay that way for long; in others, disclosure of secrets is unlikely. Attempting to employ secrey of design is pointless for environments where keeping secrets is infeasible. Here are two common cases.

[11] The term "security by obscurity" is used primarily by those who oppose secrecy of design.

- Military security clearances have proven quite effective in preventing the spread of classified information. Loyalty to one's country plus threats of prison are powerful inducements for keeping classified information secret. Adoption of "need to know" as the criterion for deciding who is granted initial access to classified information also helps limit its spread.

- In non-military environments, loyalty (if it exists) is probably to an employer; possible punishments for disclosures are limited to fines and, in practice, rarely imposed. Employers turn a blind eye to their employees discussing (secret) system details with professional peer groups, believing more is gained from the exchange than lost by the disclosures. Also, employees do change employment, taking with them design and implementation secrets while at the same time changing their loyalty.

Implicit in the view held by proponents of secrecy of design is also an assumption that design and implementation details are expensive to extract from artifacts available to an attacker. Again, the veracity of the assumption depends on the environment. Reverse-engineering an executable is neither difficult nor expensive given today's software tools. (There are even tools to make sense of executables produced from source code that has first been obfuscated by applying semantics-preserving transformations.) Yet there are cases where attackers are unlikely to have access to a system executable or systems that run them. One example is a system running on a well secured server that only can be reached over a network; another example is control software embedded in a physical device (e.g., a nuclear weapon) that itself is difficult for attackers to obtain.

Opponents to secrecy of design assume that releasing code and documentation for public review will bring reports of vulnerabilities. However, a system tends to attract public scrutiny only if it will be widely deployed, protect assets of some consequence, or is claimed to embody novel security functionality. Most systems do not satisfy any of those criteria, and they will be largely ignored by reviewers looking to maximize the impact of their efforts. For example, open source software is claimed to benefit from on-going review by a large developer community[12] but most of these developers are looking to extend or change the system's functionality rather than searching through the code base for vulnerabilities. Perhaps more to the point, there is no hard evidence of fewer vulnerabilities in open source software.

There is also a question about whether vulnerabilities that are discovered by public review will be reported. Some reviewers are motivated by the publicity their discoveries will bring, and they can be expected to report (albeit in the press, which might mean negative publicity for a developer) vulnerabilities they discover. Others, however, are more motivated by what can be gained from exploiting what they discover—they will remain silent. Examples here range from individuals seeking riches by attacking financial institutions to govern-

[12]This view underlies what Eric Raymond names Linus' Law: "Given enough eyeballs, all bugs are shallow."

ments stockpiling arsenals of attacks that destabilize their opponent's critical infrastructures.

One final assumption is implicit in the opponent view to secrecy of design. It concerns the feasibility of creating and disseminating repairs once a vulnerability has been identified. Some vulnerabilities cannot be repaired by making incremental changes to already deployed systems. Even when incremental repairs are possible, some deployed systems might not be easily reached to notify about the vulnerabilities or make those repairs. In either case, public knowledge of vulnerabilities leads to an overall reduction in security by exposing to a broader community new opportunities for attack.

Merits of Keeping Known Vulnerabilities Secret. Among the system details that might be kept secret are the known vulnerabilities. Systems are routinely shipped with known vulnerabilities—the developers might believe these vulnerabilities are difficult to exploit, better addressed by adding defenses to the environment in which the system executes, or bring small risk compared to the benefits the new system offers. And after a system has been operating in the field, additional vulnerabilities are likely to become known, because they are discovered by developers or others. The obvious question in connection with secrecy of design is: Should these vulnerabilities be kept secret?

Secrecy of design proponents would argue that revealing vulnerabilities is unwise, because it facilitates attacks. Even if patches are made available immediately, these patches are probably not going to be applied to all systems right away.[13] Some systems will be performing tasks that cannot be interrupted and, therefore, a patch cannot be applied as soon as it becomes available. In other cases, operators are (justifiably) fearful that applying a patch could be destabilizing, so they undertake a period of local off-line testing before installing the patch in their production environments. In summary, public disclosure of vulnerabilities here leads to increased numbers of attacks and systems compromised.

Secrecy of design opponents hold that keeping vulnerabilities secret is a mistake. They contend that by failing to disclose the existence of vulnerabilities, a software or service provider is now guilty of misrepresenting the system's security. This is bad business, destroys customer confidence, and might even be considered a fraudulent misrepresentation with legal consequences. Furthermore, keeping a vulnerability secret from the operators of a system in no way guarantees that vulnerability will stay secret from attackers. So system owners, the party most able to institute changes in system usage as a way to compensate for a new exposure, are unable to take action due to ignorance about the new vulnerability.

[13] The experience of Microsoft is instructive. They typically observed a sharp rise in attacks just after (not before!) issuing a patch for Windows software. Apparently, attackers reverse-engineer each new patch to find the vulnerability and then devise a corresponding attack for use in compromising unpatched systems.

1.5 Real World Physical Security

Although computer security is a relatively new discipline, security for physical artifacts has been studied for centuries. Concerns about confidentiality, integrity, and availability existed long before the advent of digital computers, as did questions about dealing with flaws in defenses and deciding how best to manage risk. It is unwise to ignore these insights, although the differences between physical artifacts and digital ones must be taken into account. For example, the relative ease with which bits can be copied or transported as compared with physical objects is significant when translating security lessons from the physical world to the electronic one.

1.5.1 Security through Accountability

An attacker need find only one exploitable vulnerability, whereas the defender must be concerned with all. That asymmetry implies the chances are good that some attackers might well succeed in circumventing a system's defenses. Defenders are thus better off if the system's defenses are not the sole reason that threats are dissuaded from launching attacks.

To understand how this might work, consider how banks dissuade thieves from commiting robbery. The valuables are locked in a vault, which is difficult (but not impossible) to penetrate. There is an alarm system to alert the police when a heist is in progress. And surveillance cameras provide images to help in apprehending the burglars and evidence to support conviction.

The rational burglar (and admittedly not all burglars are) decides whether to undertake a given robbery by understanding not only how much could be gained from fencing the stolen goods and what are the chances of penetrating the vault but also what is the probability and cost of being apprehended, convicted, and punished. Effectiveness by the police in catching and the courts in convicting a burglar thus creates a disincentive to committing the crime. Surveillance cameras—not the locked vault—play the crucial role here (although aficionados of heist flicks know that time spent circumventing alarms and breaking into a vault increases a burglar's chances of being caught in the act).

Turning now to computer systems, this same structure can be obtained through Complete Mediation and three basic classes of mechanisms:[14]

> **Authorization.** An *authorization mechanism* governs whether requested actions are allowed to proceed. []
>
> **Authentication.** An *authentication mechanism* associates a principal and perhaps those it speaks for with actions or communications. []
>
> **Audit.** An *audit mechanism* records system activity, attributing each action to some responsible principal. []

[14] Authorization, Authentication, and Audit are together known as the "gold standard" for computer security because Au is the atomic symbol for gold and each of these terms starts with that prefix.

The vault is an authorization mechanism, because it regulates access; the vault's key (or combination) is an authentication mechanism, because it identifies principals who are permitted access; and the bank's surveillance cameras are an audit mechanism, because they record activity by each principal.

Note the central role that authentication plays in both authorization and audit. Humans, computers, and channels, differ in their computational and information storage capabilities, so different authentication mechanisms are typically best for each. Authentication turns out to be a rich area to explore and will be a recurring theme in this text.

We saw above that authorization is not the true disincentive for burglars to undertake a robbery; it also need not be the true disincentive for attackers attempting to subvert a computer system. Any system that supports the following has the information needed for apprehending attackers and convicting them in court, hence can dissuade threats from launching attacks.

> **Accountability.** Hold people legally responsible for actions they instigate. Employ an audit mechanism to ascertain, capture, and preserve in some irrefutable manner the association between each action undertaken and the person who is legally responsible for causing that action. []

Accountability constitutes the ultimate deterrent, whereas authorization mechanisms merely increase the chances an attack will fail or the attacker will be caught.

Supporting accountability can be tricky, though. The principal making a request is not always the principal—or even acting on behalf of the principal—that should be held accountable. For example, a program or a computer is a principal that might, by design, act on behalf of many users (i.e., principals). Which user should be held responsible for an inappropriate action by that program or computer? Add attackers to the picture, and it becomes possible for a compromised system to act under control of an attacker but appear to be acting on behalf of a *bona fide* user. Thus, enforcing accountability is not simply a matter of authenticating the source of request messages and employing an audit mechanism to archive that information.

Accountability, for all its virtues, is not appropriate in all settings. A principal's anonymity is sometimes vital for the success of an enterprise:

- Only when each vote cast is anonymous can we be certain it reflects the will of the voter. Anonymity protects the voter from retaliation for making what somebody else views as the wrong choice; it also deprives the voter of a token to justify compensation for making what somebody else thinks is the right choice.

- Critical remarks and other unpopular communication might go unsaid if the speaker's identity is known so retribution would be possible. However, anonymity here is a double-edged sword—some might be more inclined to make irresponsible, inaccurate, or incendiary statements when their identity cannot be known.

- Accountability has a chilling effect on seeking certain information. To show interest in particular diseases, political or social causes, and even technologies (e.g., chem/bio warfare or home-made explosives), for example, risks disgrace or investigation. One might be less inclined to visit a web site hosting such content if that act could become known to government, management, co-workers, family members, or (in the case of public figures) the press.

- Knowledge of who is making the offer to buy some article can benefit a seller trying to decide whether to hold out for a higher price, because if the buyer is known to have substantial resources, then the seller might be more inclined to continue negotiating. Similarly, knowledge of who is the seller, which might lead to information about the seller's circumstances, could give an edge to buyers who might exploit a seller's temporary liquidity crisis.

Governments, by and large, favor accountability—it facilitates prosecuting offenders and, by employing authentication that includes location information, it can eliminate questions of jurisdiction.[15] Business too favors a climate of accountability, since it allows partners to be chosen based on evidence of a history of successful interactions. However, as illustrated above, accountability is not a panacea. So in the final analysis, one must weigh the added security that accountability (through deterrence) provides against any adverse effects it brings to an enterprise.

1.5.2 Risk Management

Most of us practice *risk management* when contemplating how to protect valuable articles we own from theft. We pursue *risk reduction*, investing in security measures (e.g., locks and alarms) to deter burglars; we *transfer risk* by purchasing insurance; or we do both if decreasing the expected loss[16] to an acceptable level using risk reduction alone would be too expensive.

Expected loss is proportional to the value of what is being protected and to the hostility of the environment; it is inversely proportional to the efficacy of any deployed risk reduction measures. This explains why a local jewelry store likely would have stronger locks than a residence; why apartments in a big city have doors with multiple locks and windows with steel grates, while apartments in a relatively crime-free small towns have neither; and why insurance rates for a residence go down after a burglar alarm is installed. It's not just common sense to spend more for securing items of greater value and for more hostile environments—it's a straightforward consequence of risk management.

[15] Different laws apply in different locations. For example, gambling is legal in some places but not others. For accountability to be useful in prosecuting a crime, the jurisdiction must be known so that correct laws can be applied and an appropriate court used for the trial.

[16] The expected loss from theft for an article is the product of the probability the article will be stolen times the value of the article.

To practice risk management, we must know the cost of implementing various risk reduction measures, their efficacy, the value of what is being protected, and the probabilities of incurring a loss. Most of these quantities are difficult to determine with much precision. For example, in securing a residence against burglary, there are costs for choosing, purchasing, and installing multiple different locks on the doors plus a cost from the inconvenience of carrying and using multiple keys every time you enter or exit.[17] Yet despite our ignorance about exact values for the quantities involved, we can usually use the risk management framework to help make sensible, if not optimal, judgements about what security is worth deploying in various circumstances.

The practice of risk management is not limited to judgements about physical security. It also can be a useful framework when contemplating computer security. For one thing, it forces you to think in terms of expected losses, not just the costs of losses. This means you must (i) estimate a value for the loss incurred for violating the various confidentiality, integrity, and availability properties required by the system, and (ii) estimate a probability for each kind of compromise given the anticipated threat. Even very rough estimates here help avoid the temptation of deploying security that is far stronger than needed. Of course, there are settings where virtually no risk of loss is tolerable—national security comes to mind. Even here, though, the risk management framework will make this obvious.

A key parameter for risk reduction is knowing the efficacy of each defense you intend to employ. While quantitative measures for this are unlikely to be available, it seems clear that efficacy will be correlated both with the quality of the assurance argument and with how rich is the class of policies the defense can enforce. A better assurance argument means the mechanism is less likely to have vulnerabilities; a richer class of enforceable policies means that what can be enforced is likely close to what is actually needed, reducing the chance that an attacker can exploit the difference between what is and what needs to be enforced. Establishing assurance is costly, as is building mechanisms that enforce richer collections of policies. So we conclude that achieving greater efficacy increases costs.

The cost of a defense is the cost of its design, implementation, and assurance argument plus the cost of managing it and the inconvenience its deployment imposes on users. Therefore, a defense is not sensible unless its cost is less than the value of what it protects, since otherwise suffering the loses is cheaper. The lesson to learn here is that feasible security must be simple enough to manage, not cause too much inconvenience, and simple enough to build (so it is likely to be right).

Exercises for Chapter 1

1.1 Cost and delay make it impractical to search each airline passenger completely prior to every flight. Instead, sampling is employed and only a subset

[17] Be wary of ignoring the cost of inconvenience. Although this cost is difficult to quantify, it often dominates as risk reduction measures with ever higher efficacies are deployed.

of passengers are thoroughly searched. Which of the following criteria should be more effective at decreasing the chances that passengers will carry concealed weapons onto flights? Justify your answer.

i. Select passengers at random for screening. Thus, babies, grandmothers, and government officials might well be selected for search.

ii. Select for screening randomly among passengers satisfying a predefined profile. For example, based on past airline hijackings, a plausible profile might be males of a certain age and ethnicity (but feel free to propose another).

1.2 Classify each of the following as a violation of confidentiality, integrity, availability, or of some combination (and state what that is).

(a) During the final examination, Alice copies an answer from another student's paper, then realizes that answer is wrong and corrects it before submitting her paper for grading.

(b) Bob registers the domain name `AddisonWesley.com` and refuses to let the publisher Addison Wesley buy or use that domain name.

(c) Carol attempts to login to Dave's account, unsuccessfully guessing various passwords until the operating system locks the account to prevent further guessing (but also preventing Dave from logging in).

(d) Edward figures out a way to access any file on the University computer and runs a program that lowers the grades of some students he saw cheating earlier in the semester.

(e) Fran figures out a way to access any file on the University computer and runs a program that computes and reports to her the average homework grade of students in her security course.

(f) George uses an extension to listen-in on her brother's telephone conversation and accidentally forgets to hang-up the phone when he is done listening.

1.3 What kind of security property is each of the following?

(a) The grade for the assigment is available only to the student who submitted that assignment.

(b) If your course grade changed, then the professor made that change.

(c) The output is produced by the CS Department web server.

(d) Requests to the web server are not processed out of order.

(e) No run-time exception is raised during execution.

(f) User Alice may not issue read operations to file F.

(g) The program Alice runs to issue read operations on file F runs to completion.

(h) If Alice sends a piece of email then there is no way for her to deny having done so.

(i) The downloaded piece of music may be played at most 5 times.

(j) The memo may be forwarded to your employees but they may not forward it any further.

1.4 Consider the following protocol for conducting an election.

1. A set of identical paper ballots is printed. Each ballot contains the same list of candidates.
2. Each qualified voter is given a single unmarked ballot.
3. In private, the voter uses a pen to circle one name on the list and folds the ballot in half (hiding from view the list of candidates and the one that was selected).
4. The voter then places that marked ballot in the locked collection box.
5. After everyone has voted, the collection box is opened, the ballots are unfolded, counted, and a winner is announced.

(a) What properties should be satisfied by any reasonable protocol for conducting an election (and not just by the protocol outlined above)?

(b) Establish the necessity of each protocol step above by explaining how each contributes to one or more of the properties you listed in part (a).

1.5 A host and guests are dining at a fancy restaurant, where they are served by a waiter. "*In vino veritas*" (Plato), so the host decides to purchase a bottle of wine to complement the meal. The protocol for purchasing that bottle in such circumstances typically involves the following steps:

1. The host tells the waiter the name of a bottle of wine.
2. The waiter brings to the table an unopened bottle with that name on the label.
3. In the presence of the host, the waiter breaks the seal on the bottle, removes the cork, and pours a small amount into the host's glass.
4. The host samples the wine in that glass.
5. If the host finds the wine is not spoiled then the host nods approval, and the waiter pours the wine into the guests' glasses, then fills the host's glass, and leaves the bottle on the table.

What properties is this protocol designed to enforce? Explain the connection each protocol step has to these properties.

1.6 A long wine list can be intimidating, but it virtually guarantees (assume this, anyway) that the restaurant will have a suitable wine no matter what meals a host and guests at a given table order. The host who knows little about matching wines to food and who has a limited budget might engage in the following protocol.

1. Only the host is given a copy of the wine list. This list contains the price for each wine the restaurant sells.
2. The host identifies two wines that span the price range defined by the host's wine budget. Let's call them W_{low} and W_{high}.
3. After the guests select and order their meals, the host asks the waiter which of wines W_{low} and W_{high} might be most suitable for what was ordered.
4. The waiter responds with a list of suggestions, where the waiter's suggestions are priced between the prices of W_{low} and W_{high}, and each suggestion is also well matched to all the food that has been ordered. (The waiter's suggestions might or might not include wines W_{low} and W_{high}.)
5. The host orders one of the wines the waiter suggested.

What properties is this protocol designed to enforce? Explain the connection each protocol step has to these properties.

1.7 Here is the usual protocol for using a credit card to pay for dinner in a restaurant.

1. The waiter gives the bill to the host.
2. The host looks over the bill and, if all seems correct, hands a credit card to the waiter.
3. The waiter returns with the credit card and two copies of a credit card charge slip. Each copy lists the amount on the bill.
4. The host looks at the charge slips, adds a gratuity (if desired) onto one copy, and signs that charge slip. The host keeps the credit card and the other copy of the charge slip.

Consider a different protocol:

1. The host gives a credit card to waiter.
2. The waiter returns with the bill, the credit card, and two copies of a credit card charge slip. Each copy lists the amount on the bill.
3. The host looks at the charge slips, adds a gratuity (if desired) onto one copy, and signs that charge slip.
4. The host keeps the credit card and the other copy of the charge slip.

(a) The two protocols exhibit performance differences, but do they otherwise satisfy the same properties? If they do not satisfy the same properties then what are the differences?

(b) What assumptions about expected-case behavior underlie each protocol, and what are the performance implications when that expected-case behavior does not hold?

1.8 Consider an enlightened company, where employees who have free time may use their office computers to access the Internet for personal tasks. A newspaper article causes management to fear that the company's secret documents are being leaked to the press, and that prompts an audit to identify which employees have electronic copies of secret documents. To implement that audit, the security officer proposes that a virus be written and used to infect all machines on the company's intranet. That virus would behave as follows.

1. This virus periodically scans the disk of any machine it infects, locating any secret documents being stored there.
2. Whenever the virus locates a secret document, it sends email containing the name of the machine and secret document to the security officer.

Discuss whether this scheme violates employee privacy.

1.9 Suppose the virus in exercise 1.8 worked somewhat differently. Instead of reporting all secret documents found, it simply reports the name of every document found that is not on an approved list of publicly-released corporate memos. Do you believe this violates employee privacy? Explain why.

1.10 Indicate, for each of the following, the extent to which user privacy is being violated.

(a) When the user's browser opens a web page being hosted by an Internet portal (such as Google, MSN, or Yahoo), a pop-up appears containing an advertisement selected based on the last web search that user made.

(b) When the user's browser opens a web page being hosted by an Internet portal (such as Google, MSN, or Yahoo), a pop-up appears containing an advertisement selected based on the contents of the last email that user read or sent.

1.11 Must confidentiaility be sacrificed to achieve accountability? Propose a scheme whereby confidentiality is sacrificed only when a principal is being accused of violating a rule or law.

1.12 A relation $R(x, y)$ is defined to be *reflexive* iff $R(x, x)$ always holds, *symmetric* iff whenever $R(x, y)$ holds then so does $R(y, x)$, and *transitive* if whenever both $R(x, y)$ and $R(y, z)$ hold then so does $R(x, z)$.

(a) Consider relation $trusts(c, c')$, which holds iff component c trusts component c'. Should we necessarily expect *trusts* to be reflexive, symmetric, or transitive? Explain, giving examples to support your views.

(b) Consider the relation $tw(p, p')$, which holds iff principal p believes that principal p' is trustworthy. Should we necessarily expect tw to be reflexive, symmetric, or transitive? Explain, giving examples to support your views.

1.13 In §1.4 attacks are equated with instruction execution:

> Successfully attacking a computer causes the target to execute instructions that it shouldn't, resulting in violation of some security property.

Discuss execution that might be involved in the following kinds of attacks.

(a) Violating a confidentaility property that involves content on a disk.

(b) Violating an availability property concerning query processing for an in-memory database.

(c) A buffer-overflow attack.

(d) Violating an integrity property that involves a downloaded piece of music being played more times than permitted by the license agreement.

1.14 Access control lists are often used by authorization mechanisms that enforce confidentiality and integrity of files. The access control list ACL_F is stored in the directory that contains the file F being governed by ACL_F. The integrity of that directory must be protected, and this is accomplished using an access control list stored in yet another directory, and so on. It would seem there is now an infinite-regress. Explain how this infinite regress might be avoided.

1.15 Consider the following exhortation:

> **Principal of Least Privilege.** Each task should be assigned to the principal that
>
> i. has the least set of privileges and
>
> ii. is capable of accomplishing the task.
>
> []

(a) In what sense is this an instance of the Principle of Least Privilege?

(b) In what sense are the two different? Illustrate by examples.

1.16 Compare and contrast the Principle of Least Privilege and the military's "Need to know" principle for allowing access to confidential information.

1.17 A medieval castles was often situated in a large open field, and the castle itself was surrounded by high stone walls. Skilled archers perched on top of those walls, and outside of the walls was a moat.

(a) Discuss the extent to which these defenses satisfy the independence requirement we require for an effective defense in depth.

(b) Discuss the extent to which these defense satisfy the overlap requirement we require for an effective defense in depth.

(c) Identify points of similarity shared by these mechanisms and, for each, outline an attack.

1.18 Passwords and cryptographic keys are not effective unless they are kept secret. Reconcile this requirement with the exhortation to avoid security by obscurity.

1.19 Machine guns and land mines can be deployed as a perimeter defense against infantry attacks, just as a firewall can serve as a perimeter defense against malware from the Internet. In both cases, action by the perimeter defense stops the adversary from advancing to the interior.

(a) Discuss whether secrecy of design applies to informing the adversary about the exact location of the machine guns and land mines.

(b) Discuss whether secrecy of design applies to informing the adversay about the exact criteria used by the firewall for deciding whether or not to block a packet.

(c) What are the essential similarities and differences for these two perimeter defenses that would explain the similar/different conclusions you give for parts (a) and (b).

1.20 Suppose sampling is being used to select the subset of air travelers subject to a detailed security search, and this sampling is based on a secret profile.

(a) Discuss whether having this profile be secret is secrecy of design and whether the decision to keep the profile secret leads to better security.

(b) Suppose the sampling is based on a secret list of travellers who might be terrorists. Discuss whether keeping secret the criteria for membership on this list of names is secrecy of design and whether that decision to keep that secret leads to better security.

(c) Discuss whether keeping secret the list of names itself in part (b) is secrecy of design and whether that decision to keep that secret leads to better security.

Notes and Reading

Many terms have both been used to mean continued and correct system operation in settings where mother nature and malevolent actors cannot be ignored. With the 1999 publication of the National Academy's *Trust in Cyberspace* [16] and Microsoft's 2002 Trustworthy Computing initiative to revamp its Windows software, the term "trustworthiness" has emerged as the more popular locution.

That said, the computer security research community has until recently been quite independent of the somewhat older fault-tolerance research community, even though interpreting dependability to include security had been advocated within the latter [10]. Such insularity was problematic because, as illustrated at the start of this chapter in the discussion about replication and confidentiality for Byzantine fault-tolerance, security cannot be ignored when implementing fault-tolerance and *vice versa*. Byzantine fault-tolerance, by the way, was developed in connection with the SIFT (Software Implemented Fault Tolerance) project [21] at SRI to build a computing system for control of fly-by-wire aircraft, although the term "Byzantine faults" doesn't appear in print until a 1982 paper [8], written to popularize an agreement algorithm developed for SIFT.

We distinguish the terms threat, vulnerability, and attack, in this book because doing so creates a richer language and enables more precise technical discussions. Not all authors make these distinctions. However, there is universal agreement on informal definitions for confidentiality (or secrecy), integrity, and availability, and most authors agree that these are the foundation for what we are calling security policies (though the distinction between "policies" and "properties" we are making in §1.2 is not widespread). Our definition of integrity is based on the formal definition in [1] of safety properties [7] and our definition of availability is based on the formal defniition in [2] of liveness properties [7]. The distinction between confidentiality properties and safety properties was first noted by McLean in [12], and there is still no universally accepted formal definition of confidentiality.

The disturbing revelation in §1.2.1 that gender, date of birth, and zip code together constitute a unique identifier for people in Cambridge, Massachusets is described by Latanya Sweeney [17] to illustrate privacy violations allowed by certain so-called anonymous databases. And the insight we give at the end of §1.2.1 about using context to define privacy comes from a philosopher, Helen Nissembaum, who developed it while working with computer security researchers; her approach is called *contextual integrity* [13]. See [20] for an excellent survey and snapshot of contemporary thinking about technical and legal issues concerned with privacy.

The various principles we give in §1.3 and §1.4 are derived from a paper by Saltzer and Schroeder [15] which, though written in the mid 1970's, remains worth reading today. An account of some devious ways that trust can be misplaced is the basis for the 1983 Turing Award acceptance speech [19] delivered by Ken Thompson. That security mechanisms be viewed through the lens of isolation, monitoring, and recovery is suggested by Lampson in [9]. In depth discussions of the various security mechanisms appear throughout this book,

and citations to additional reading are given then.

Much has been written about secrecy of design, though one frequently finds newer papers simply repeat old arguments. Our discussion of design secrecy is based on the one appearing in Appendix I of [16]. However, the idea that security of a mechanism should not depend on the secrecy of its design is generally credited to the Dutch cryptographer Auguste Kerckhoffs, who proposed in an 1883 essay [6] that the security of a military cryptosystem depend on keeping the key secret but not the design. Eric Raymond's essay "The Cathedral and the Bazaar" in [14] is the classic description of open software, and it is recommended reading for understanding this truly innovative approach to software development. Finally, a piece of fresh thinking on open design is the article [18] by Peter Swire, which characterizes when disclosure can help enhance security (and when it does not) by comparing the stipulation that "there is no security by obscurity" to the World War II admonition "loose lips sink ships".

Lampson's argument in [9] provided the inspiration for our discussion in §1.5.1 that security should be based on accountability. Also the "gold standard" (authorization, authentication, and audit) is described in [9], although Lampson started using this mnemonic in Fall 2000. See [5] for a more complete discussion of the benefits and risks associated with accountability as the basis for enforcement, and read Larry Lessig's seminal opus [11] for an extensive discussion about legal and societal issues that come with universally embracing accountability.

Risk management has its roots in the business world—economics and, in particular, investment. Many have advocated risk management for computer security, with Ross Anderson among the more effective spokesman by first portraying security as an engineering enterprise in his textbook [4] and then, with papers like [3], helping to establish "security economics" as a respectable area of inquiry.

Bibliography

[1] Bowen Alpern, Alan J. Demers, and Fred B. Schneider. Safety without stuttering. *Information Processing Letters*, 23(4):177–180, November 1986.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] Ross Anderson and Tyler Moore. The economics of information security. *Science*, 314(5799):610–613, October 2006.

[4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.

[5] Seymour Goodman and Herbert Lin, editors. *Towards a Safer and More Secure Cyberspace*. National Academy Press, 2007.

[6] Auguste Kerchhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX:5–38, January 1883.

[7] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[9] Butler Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, June 2004.

[10] J. C. LaPrie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[11] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, 2000.

[12] John McLean. A general theory of composition of trace sets closed under selective interleaving functions. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 79–93, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[13] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 79(1):119–158, 2004.

[14] Eric S. Raymond. *Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, 2001.

[15] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, March 1975.

[16] Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1999.

[17] Latanya Sweeney. k-Anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[18] Peter Swire. A model for when disclosure helps security: What is different about computer and network security? *Journal on Telecommunications and High Technology Law*, 3(1):163–208, 2004.

[19] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[20] Jim Waldo, Herbert Lin, and Lynette Millett, editors. *Engaging Privacy and Information Technology in a Digital Age*. National Academy Press, 2007.

[21] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P.M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.