Chapter 1

Basic Concepts and Principles

1.1 Fundamental goals of computer security 2
1.2 Computer security policies and attacks 4
1.3 Risk, risk assessment, and modeling expected losses 6
1.4 Adversary modeling and security analysis 9
1.5 Threat modeling: diagrams, trees, lists and STRIDE 11
1.6 Model-reality gaps and real-world outcomes 16
1.7 ‡Design principles for computer security 20
1.8 ‡Why computer security is hard 25
1.9 ‡End notes and further reading 27
References 28

The official version of this book is available at
`https://www.springer.com/gp/book/9783030336486`

ISBN: 978-3-030-33648-6 (hardcopy), 978-3-030-33649-3 (eBook)

version: 25 Sept 2019

Chapter 1

Basic Concepts and Principles

Our subject area is computer and Internet security—the security of software, computers and computer networks, and of information transmitted over them and files stored on them. Here the term *computer* includes programmable computing/communications devices such as a personal computer or mobile device (e.g., laptop, tablet, smartphone), and machines they communicate with including servers and network devices. Servers include front-end servers that host web sites, back-end servers that contain databases, and intermediary nodes for storing or forwarding information such as email, text messages, voice, and video content. Network devices include firewalls, routers and switches. Our interests include the software on such machines, the communications links between them, how people interact with them, and how they can be misused by various agents.

We first consider the primary objectives or *fundamental goals* of computer security. Many of these can be viewed as *security services* provided to users and other system components. Later in this chapter we consider a longer list of *design principles* for security, useful in building systems that deliver such services.

1.1 Fundamental goals of computer security

We define *computer security* as the combined art, science and engineering practice of protecting computer-related assets from unauthorized actions and their consequences, either by preventing such actions or detecting and then recovering from them. Computer security aims to protect data, computer hardware and software plus related communications networks, and physical-world devices and elements they control, from *intentional* misuse by unauthorized parties—i.e., access or control by entities other than the legitimate owners or their authorized agents. Mechanisms protecting computers against *unintentional* damage or mistakes, or that fall under the categories of *reliability* and *redundancy*, are also at times essential for overall computer security, but are not our main focus herein.

The overall goal of computer security is to support computer-based services by providing security properties that help deliver on expectations. We begin by discussing the main properties of interest.

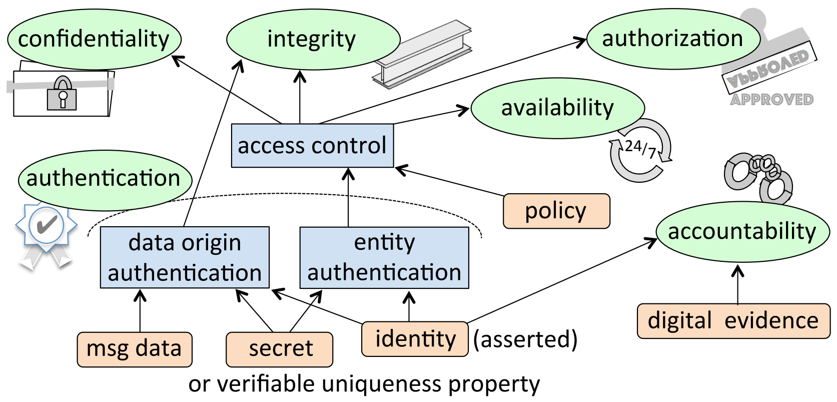


Figure 1.1: Six high-level computer security goals (properties delivered as a service). Icons denote end-goals. Important supporting mechanisms are shown in rectangles.

1) *confidentiality*: the property of non-public information remaining accessible only to authorized parties, whether stored (at rest) or in transit (in motion). This is supported by access control (below), including mechanisms enforced by an operating system. A common *technical* means, *data encryption*, involves keyed cryptographic algorithms; access to a secret key allows recovery of meaningful information from encrypted data. Confidentiality can also be provided by *procedural* means, e.g., by allowing offline storage media to be physically accessed only by authorized individuals.

2) *integrity*: the property of data, software or hardware remaining unaltered, except by authorized parties. While error detection and *error correction codes* address some benign errors (including in hardware), access controls and *cryptographic checksums* are used to combat malicious integrity violations. The *integrity* of people (to resist bribery, blackmail, coercion) is a different use of this word, but is related and important.

3) *authorization* (authorized access): the property of computing resources being accessible only by authorized entities, e.g., those approved by the resource owner or domain administrator. Authorized access is achieved through *access control* mechanisms, which restrict access to physical devices, software services, and information.

4) *availability*: the property of information, services and computing resources remaining accessible for authorized use. Aside from reliable hardware and software, this requires protection from intentional deletion and disruption, including *denial of service attacks* aiming to overwhelm resources.

In discussing security, the agents representing users, communicating entities, or system processes are called *principals*. A principal has associated *privileges* specifying the resources it is authorized to access. The identity of a principal is thus important—but asserted identities must be verified. This leads to the following two further goals.

5) *authentication*: assurance that a principal, data, or software is genuine relative to expectations arising from appearances or context. *Entity authentication* provides assurances that the identity of a principal involved in a transaction is as asserted; this sup-

ports authorization (above). *Data origin authentication* provides assurances that the source of data or software is as asserted; it also implies data integrity (above). Note that data modification by an entity other than the original source changes the source. Authentication supports *attribution*—indicating to whom an action can be ascribed—and thus accountability.

6) *accountability*: the ability to identify principals responsible for past actions. As the electronic world lacks conventional evidence (e.g., paper trails, human memory of observed events), accountability is achieved by transaction evidence or logs recorded by electronic means, including identifiers of principals involved, such that principals cannot later credibly deny (*repudiate*) previous commitments or actions.

TRUSTED VS. TRUSTWORTHY. We carefully distinguish the terms *trusted* and *trustworthy* as follows. Something is trustworthy if it *deserves* our confidence, i.e., will reliably meet expectations. Something trusted *has* our confidence, whether deserved or not; so a trusted component is relied on to meet expectations, but if it fails then all guarantees are lost. For example, if we put money in a bank, we trust the bank to return it. A bank that has, over 500 years, never failed to return deposits would be considered trustworthy; one that goes bankrupt after ten years may have been trusted, but was not trustworthy.

CONFIDENTIALITY VS. PRIVACY, AND ANONYMITY. Confidentiality involves information protection to prevent unauthorized disclosure. A related term, *privacy* (or *information privacy*), more narrowly involves *personally sensitive* information, protecting it, and controlling how it is shared. An individual may suffer anxiety, distress, reputational damage, discrimination, or other harm upon release of their home or email address, phone number, health or personal tax information, political views, religion, sexual orientation, consumer habits, or social acquaintances. What information should be private is often a personal choice, depending on what an individual desires to selectively release. In some cases, privacy is related to *anonymity*—the property that one's actions or involvement are not linkable to a public identity. While neither privacy nor anonymity is a main focus of this book, many of the security mechanisms we discuss are essential to support both.

1.2 Computer security policies and attacks

Consider the statement: *This computer is secure.* Would you and a friend independently write down the same thing if asked to explain what this means? Unlikely. How about: *This network is secure. This web site is secure. This protocol is provably secure.* Again, unlikely. To remove ambiguity—ambiguity is security's enemy—we need more precise definitions, and a richer vocabulary of security-specific terminology.

ASSETS, POLICY. Computer security protects resources or *assets*: information, software, hardware, and computing and communications services. Computer-based data manipulation allows control of many physical-world resources such as financial assets, physical property, and infrastructure. Security is formally defined relative to a *security policy*, which specifies the design intent of a system's rules and practices—what is, and is not (supposed to be) allowed. The policy may explicitly specify assets requiring protection;

specific users allowed to access specific assets, and the allowed means of access;[1] security services to be provided; and system controls that must be in place. Ideally, a system enforces the rules implied by its policy. Depending on viewpoint and methodology, the policy either dictates, or is derived from, the system's security requirements.

THEORY, PRACTICE. In theory, a formal security policy precisely defines each possible system state as either authorized (*secure*) or unauthorized (*non-secure*). Non-secure states may bring harm to assets. The system should start in a secure state. System actions (e.g., related to input/output, data transfer, or accessing ports) cause state transitions. A security policy is *violated* if the system moves into an unauthorized state. In practice, security policies are often informal documents including guidelines and expectations related to known security issues. Formulating precise policies is more difficult and time-consuming. Their value is typically under-appreciated until security incidents occur. Nonetheless, security is defined relative to a policy, ideally in written form.

ATTACKS, AGENTS. An *attack* is the deliberate execution of one of more steps intended to cause a *security violation*, such as unauthorized control of a client device. Attacks exploit specific system characteristics called *vulnerabilities*, including design flaws, implementation flaws, and deployment or configuration issues (e.g., lack of physical isolation, ongoing use of known default passwords, debugging interfaces left enabled). The source or *threat agent* behind a potential attack is called an *adversary*, and often called an *attacker* once a threat is activated into an actual attack. Figure 1.2 illustrates these terms.

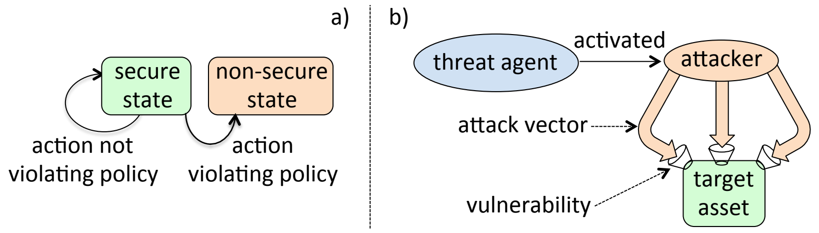


Figure 1.2: Security policy violations and attacks. a) A policy violation results in a non-secure state. b) A threat agent becomes active by launching an attack, aiming to exploit a vulnerability through a particular attack vector.

THREAT. A *threat* is any combination of circumstances and entities that might harm assets, i.e., cause security violations. A credible threat has both capable means and intentions. The mere existence of a threat agent and a vulnerability that they have the capability to exploit on a target system does not necessarily imply that an attack will be instantiated in a given time period; the agent may fail to take action, e.g., due to indifference or insufficient incentive. Computer security aims to protect assets by mitigating threats, largely by identifying and eliminating vulnerabilities, thereby disabling viable *attack vectors*—specific methods, or sequences of steps, by which attacks are carried out. Attacks typically have specific objectives, such as: extraction of strategic or personal information;

[1]For example, corporate policy may allow authorized employees remote access to regular user accounts via SSH (Chapter 10), but not remote access to a *superuser* or *root* account. A password policy (Chapter 3) may require that passwords have at least 10 characters including two non-alphabetic characters.

disruption of the integrity of data or software (including installation of rogue programs); remotely harnessing a resource, such as malicious control of a computer; or *denial of service*, resulting in blocked access to system resources by authorized users. Threat agents and attack vectors raise the questions: secure against whom, from what types of attacks?

CONTROLS. A security policy helps in determining when a security violation has occurred, but by itself does not preclude such violations. To support and enforce security policies—that is, to prevent violations, or detect violations in order to react to limit damage, and recover—*controls* and *countermeasures* are needed. These include operational and management processes, operating system enforcement by software *monitors* and related access control measures, and other *security mechanisms*—technical means of enforcement involving specialized devices, software techniques, algorithms or protocols.

Example *(House security policy).* To illustrate this terminology with a non-technical example, consider a simple security policy for a house: no one is allowed in the house unless accompanied by a family member, and only family members are authorized to remove physical objects from the house. An unaccompanied stranger in the house is a security violation. An unlocked back door is a vulnerability. A stranger (attacker) entering through such a door, and removing a television, amounts to an attack. The attack vector is entry through the unlocked door. A threat here is the existence of an individual motivated to profit by stealing an asset and selling it for cash.

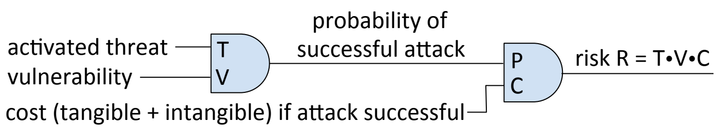
1.3 Risk, risk assessment, and modeling expected losses

Most large organizations are obligated, interested, or advised to understand the losses that might result from security violations. This leads to various definitions of *risk* and approaches to *risk assessment*. In our context, we define *risk* as the expected loss due to harmful future events, relative to an implied set of assets and over a fixed time period. Risk depends on threat agents, the probability of an attack (and of its success, which requires vulnerabilities), and expected losses in that case. *Risk assessment* involves analyzing these factors in order to estimate risk. The idealistic goal of *quantitative risk assessment* is to compute numerical estimates of risk; however for reasons discussed below, precise such estimates are rarely possible in practice. This motivates *qualitative risk assessment*, with the more realistic goal of comparing risks relative to each other and ranking them, e.g., to allow informed decisions on how to prioritize a limited defensive budget across assets.

RISK EQUATIONS. Based on the above definition of risk, a popular *risk equation* is:

$$R = T \cdot V \cdot C \tag{1.1}$$

T reflects threat information (essentially, the probability that particular threats are instantiated by attackers in a given period). V reflects the existence of vulnerabilities. C reflects asset value, and the cost or impact of a successful attack. Equation (1.1) highlights the main elements in risk modeling and obvious relationships—e.g., risk increases with threats (and with the likelihood of attacks being launched); risk requires the presence of a vulnerability; and risk increases with the value of target assets.


activated threat
vulnerability
T
V
probability of successful attack
P
C
risk R = T•V•C
cost (tangible + intangible) if attack successful
Figure 1.3: Risk equation. Intangible costs may include corporate reputation.

Equation (1.1) may be rewritten to combine T and V into a variable P denoting the probability that a threat agent takes an action that successfully exploits a vulnerability:

$$R = P \cdot C \tag{1.2}$$

Example *(Risk due to lava flows).* Most physical assets are vulnerable to damage from hot lava flows, but the risk vanishes if there are no volcanos nearby. Equation (1.1) reflects this: even if $V = 1$ and $C = \$100$ million, the risk R equals 0 if $T = 0$. Most assets, however, are subject to other threats aside from hot lava flows.

ESTIMATING UNKNOWNS. Risk assessment requires expertise and familiarity with specific operating environments and the technologies used therein. Individual threats are best analyzed in conjunction with specific vulnerabilities exploited by associated attack vectors. The goal of producing precise quantitative estimates of risk raises many questions. How do we accurately populate parameters T, V and C? Trying to combine distinct threats into one value T is problematic—there may be countless threats to different assets, with the probabilities of individual threats depending on the agents behind each (adversary models are discussed shortly). Looking at (1.1) again, note that risk depends on combinations of threats (threat sources), vulnerabilities, and assets; for a given category of assets, the overall risk R is computed by summing across combinations of threats and vulnerabilities. A side note is that the impact or cost C relative to a given asset varies depending on the stakeholder. (For a given stakeholder, one could consider, for each asset or category of assets, the set $\mathcal{E}$ of events that may result in a security violation, and evaluate $R = R(e)$ for each $e \in \mathcal{E}$ or for disjoint subsets $E \subseteq \mathcal{E}$; this would typically require considering categories of threats or threat agents.) Indeed, computing R numerically is challenging (more on this below).

MODELING EXPECTED LOSSES. Nonetheless, to pursue quantitative estimates, noting that risk is proportional to impact per event occurrence allows a formula for *annual loss expectancy*, for a given asset:

$$ALE = \sum_{i=1}^{n} F_i \cdot C_i \tag{1.3}$$

Here the sum is over all security events modeled by index i, which may differ for different types of assets. F_i is the estimated annualized frequency of events of type i (taking into account a combination of threats, and vulnerabilities that enable threats to translate into successful attacks). C_i is the average loss expected per occurrence of an event of type i.

RISK ASSESSMENT QUESTIONS. Equations 1.1–1.3 bring focus to some questions that are fundamental not only in risk assessment, but in computer security in general:

1. What assets are most valuable, and what are their values?
2. What system vulnerabilities exist?
3. What are the relevant threat agents and attack vectors?
4. What are the associated estimates of attack probabilities, or frequencies?

COST-BENEFIT ANALYSIS. The cost of deploying security mechanisms should be accounted for. If the total cost of a new defense exceeds the anticipated benefits (e.g., lower expected losses), then the defense is unjustifiable from a *cost-benefit analysis* viewpoint. ALE estimates can inform decisions related to the cost-effectiveness of a defensive countermeasure—by comparing losses expected in its absence, to its own annualized cost.

Example *(Cost-benefit of password expiration policies).* If forcing users to change their passwords every 90 days reduces monthly company losses (from unauthorized account access) by $1000, but increases monthly help-desk costs by $2500 (from users being locked out of their accounts as a result of forgetting their new passwords), then the cost exceeds the benefit before even accounting for usability costs such as end-user time.

RISK ASSESSMENT CHALLENGES. Quantitative risk assessment may be suited to incidents that occur regularly, but not in general. Rich historical data and stable statistics are needed for useful failure probability estimates—and these exist over large samples, e.g., for human life expectancies and time-to-failure for incandescent light bulbs. But for computer security incidents, relevant such data on which to base probabilities and asset losses is both lacking and unstable, due to the infrequent nature of high-impact security incidents, and the uniqueness of environmental conditions arising from variations in host and network configurations, and in threat environments. Other barriers include:

- incomplete knowledge of vulnerabilities, worsened by rapid technology evolution;
- the difficulty of quantifying the value of intangible assets (strategic information, corporate reputation); and
- incomplete knowledge of threat agents and their *adversary classes* (Sect. 1.4). Actions of unknown intelligent human attackers cannot be accurately predicted; their existence, motivation and capabilities evolve, especially for *targeted attacks*.

Indeed for unlikely events, ALE analysis (see above) is a guessing exercise with little evidence supporting its use in practice. Yet, risk assessment exercises still offer benefits—e.g., to improve an understanding of organizational assets and encourage assigning values to them, to increase awareness of threats, and to motivate contingency and recovery planning prior to losses. The approach discussed next aims to retain the benefits while avoiding inaccurate numerical estimates.

QUALITATIVE RISK ASSESSMENT. As numerical values for threat probabilities (and impact) lack credibility, most practical risk assessments are based on *qualitative* ratings and comparative reasoning. For each asset or asset class, the relevant threats are listed; then for each such asset-threat pair, a categorical rating such as *(low, medium, high)* or perhaps ranging from *very low* to *very high*, is assigned to the probability of that threat action being launched-and-successful, and also to the impact assuming success. The combination of probability and impact rating dictates a risk rating from a combination matrix

C (cost or impact)	*P* (probability)				
	V.LOW	LOW	MODERATE	HIGH	V.HIGH
V.LOW (negligible)	1	1	1	1	1
LOW (limited)	1	2	2	2	2
MODERATE (serious)	1	2	3	3	3
HIGH (severe or catastrophic)	2	2	3	4	4
V.HIGH (multiply catastrophic)	2	3	4	5	5

Table 1.1: Risk Rating Matrix. Entries give coded risk level 1 to 5 (V.LOW to V.HIGH) as a qualitative alternative to equation (1.2). V. denotes VERY; *C* is the anticipated adverse effect (level of impact) of a successful attack; *P* is the probability that an attack both occurs (a threat is activated) and successfully exploits a vulnerability.

such as Table 1.1. In summary, each asset is identified with a set of relevant threats, and comparing the risk ratings of these threats allows a ranking indicating which threat(s) pose the greatest risk to that asset. Doing likewise across all assets allows a ranked list of risks to an organization. In turn, this suggests which assets (e.g., software applications, files, databases, client machines, servers and network devices) should receive attention ahead of others, given a limited computer security budget.

RISK MANAGEMENT VS. MITIGATION. Not all threats can (or necessarily should) be eliminated by technical means alone. *Risk management* combines the technical activity of estimating risk or simply identifying threats of major concern, and the business activity of "managing" the risk, i.e., making an informed response. Options include (a) mitigating risk by technical or procedural countermeasures; (b) transferring risk to third parties, through insurance; (c) accepting risk in the hope that doing so is less costly than (a) or (b); and (d) eliminating risk by decommissioning the system.

1.4 Adversary modeling and security analysis

An important part of any computer security analysis is building out an *adversary model*, including identifying which *adversary classes* a target system aims to defend against—a lone gunman on foot calls for different defenses than a battalion of tanks supported by a squadron of fighter planes.

ADVERSARY ATTRIBUTES. Attributes of an adversary to be considered include:

1. *objectives*—these often suggest target assets requiring special protection;
2. *methods*—e.g., the anticipated attack techniques, or types of attacks;
3. *capabilities*—computing resources (CPU, storage, bandwidth), skills, knowledge, personnel, opportunity (e.g., physical access to target machines);
4. *funding level*—this influences attacker determination, methods and capabilities; and
5. *outsider vs. insider*—an attack launched without any prior special access to the target network is an *outsider attack*. In contrast, *insiders* and *insider attacks* originate

	Named Groups of Adversaries
1	foreign intelligence (including government-funded agencies)
2	cyber-terrorists or politically-motivated adversaries
3	industrial espionage agents (perhaps funded by competitors)
4	organized crime (groups)
5	lesser criminals and *crackers*† (i.e., individuals who break into computers)
6	malicious *insiders* (including disgruntled employees)
7	non-malicious employees (often security-unaware)

Table 1.2: Named groups of adversaries. †The popular media uses the term *hackers*, which others use for computer system experts knowledgeable about low-level details.

from parties having some starting advantage, e.g., employees with physical access or network credentials as legitimate users.

The line between outsiders and insiders can be fuzzy, for example when an outsider somehow gains access to an internal machine and uses it to attack further systems.

SCHEMAS. Various schemas are used in modeling adversaries. A *categorical schema* classifies adversaries into *named groups*, as given in Table 1.2. A *capability-level schema* groups generic adversaries based on a combination of capability (opportunity and resources) and intent (motivation), say from Level 1 to 4 (weakest to strongest). This may also be used to sub-classify named groups. For example, intelligence agencies from the U.S. and China may be in Level 4, insiders could range from Level 1 to 4 based on their capabilities, and novice crackers may be in Level 1. It is also useful to distinguish *targeted attacks* aimed at specific individuals or organizations, from *opportunistic attacks* or *generic attacks* aimed at arbitrary victims. Targeted attacks may either use generic tools, or leverage *target-specific personal information*.

SECURITY EVALUATIONS AND PENETRATION TESTING. Some government departments and other organizations may require that prior to purchase or deployment, products be *certified* through a *formal security evaluation* process. This involves a third party lab reviewing, at considerable cost and time, the final form of a product or system, to verify conformance with detailed evaluation criteria as specified in relevant standards; as a complication, recertification is required once even the smallest changes are made. In contrast, self-assessments through *penetration testing* (*pen testing*) involve customers or hired consultants (with prior permission) finding vulnerabilities in deployed products by demonstrating exploits on their own live systems; interactive and automated toolsets run attack suites that pursue known design, implementation and configuration errors compiled from previous experience. Traditional pen testing is *black-box*, i.e., proceeds without use of insights from design documents or source code; use of such information (making it *white-box*) increases the chances of finding vulnerabilities and allows tighter integration with overall security analysis. Note that tests carried out by product vendors prior to product release, including common regression testing, remain important but cannot find issues arising from customer-specific configuration choices and deployment environments.

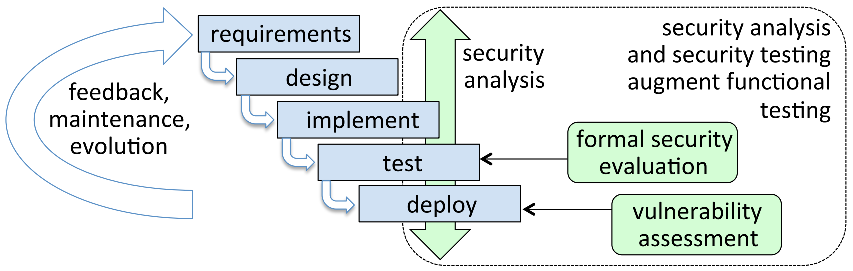


Figure 1.4: Security analysis and the software development lifecycle. The goal is to provide confidence in a system's ability to resist attacks, including by direct testing against known attacks. For a complementary view of security analysis, see Figure 1.10.

SECURITY ANALYSIS. Our main interest in *security analysis* is as in the informal phrase "carrying out a security analysis", by any methodology. The primary aim is to identify vulnerabilities related to design, and overlooked threats; analysis ideally begins early in a product's lifecycle, and continues in parallel with design and implementation (Fig. 1.4). Techniques including manual source code review and security review of design documents can uncover vulnerabilities not apparent through black-box testing alone. A secondary aim is to suggest ways to improve defenses when weaknesses are found. The term *vulnerability assessment* (Chapter 11) refers to identifying weaknesses in deployed systems, including by pen testing; security analysis, broadly viewed, includes this. Traditional security analysis considers a system's architectural features and components, identifies where protection is needed, and details how existing designs meet security requirements. Ideally this takes into account a specific target deployment environment with relevant assumptions and threats, as identified by *threat modeling* (Section 1.5), the cornerstone of security analysis. The analysis should trace how existing or planned defense mechanisms address identified threats, and note threats that remain unmitigated.

SECURITY MODELS. Security analysis may be aided by building an abstract *security model*, which relates system components to parts of a security policy to be enforced; the model may then be explored to increase confidence that system requirements are met. Such models can also be designed prior to defining policies. Security analysis benefits strongly from experience, including insights from design principles that suggest things to look for, and to avoid, in the design of security-minded products and systems.

1.5 Threat modeling: diagrams, trees, lists and STRIDE

A *threat model* identifies threats, threat agents, and attack vectors that the target system considers in scope to defend against—known from the past, or anticipated. Those considered out of scope should be explicitly recorded as such. Threat modeling takes into account *adversary modeling* (Section 1.4), and should identify and consider all *assumptions* made about the target system, environment, and attackers. Threat modeling can be done by several different approaches (Figure 1.5), as discussed in the subsections below.

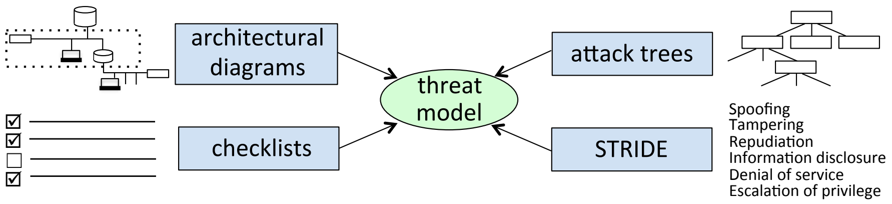


Figure 1.5: Examples of threat modeling approaches.

1.5.1 Diagram-driven threat modeling

A visual approach to threat modeling starts with an architectural representation of the system to be built or analyzed. Draw a diagram (e.g., Figure 1.6) showing target-system components and all communications links used for data flows between them. Identify and mark system gateways where system controls restrict or filter communications. Use these to delimit what might informally be called *trust domains*. For example, if users log in to a server, or the communication path is forced through a firewall gateway, draw a colored rectangle around the server and interior network components to denote that this area has different trust assumptions associated with it (e.g., users within this boundary are authenticated, or data within this boundary has passed through a filter). Now ask how your trust assumptions, or expectations of who controls what, might be violated. Focus on each component, link and domain in turn. Ask: "Where can bad things happen? How?"

Add more structure and focus to this process by turning the architectural diagram into an informal *data flow diagram*: trace the flow of data through the system for a simple task, transaction, or service. Examining this, again ask: "What could go wrong?" Then consider more complex tasks, and eventually all representative tasks.

Consider *user workflow*: trace through user actions from the time a task begins until it ends. Begin with common tasks. Move to less frequent tasks, e.g., account creation or registration (de-registration), installing, configuring and upgrading software (also abandoning, uninstalling). Consider full *lifecycles* of data, software, accounts (Figure 1.7).

Revisit your diagram and highlight where sensitive data files are stored—on servers, user devices? Double-check that all authorized access paths to this data are shown. (Are there other possibilities, e.g., access from non-standard paths? How about from backup

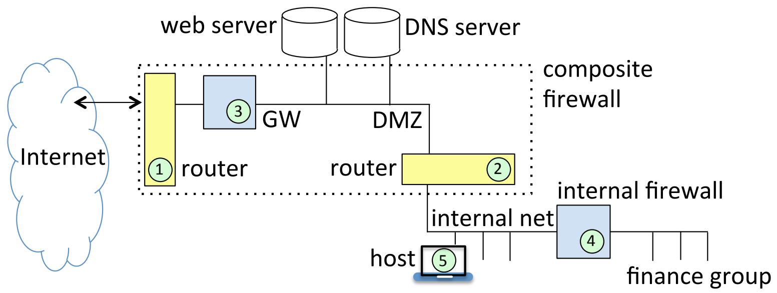


Figure 1.6: Starting point for diagram-driven threat modeling (example). This firewall architecture diagram reappears in Chapter 10, where its components are explained.

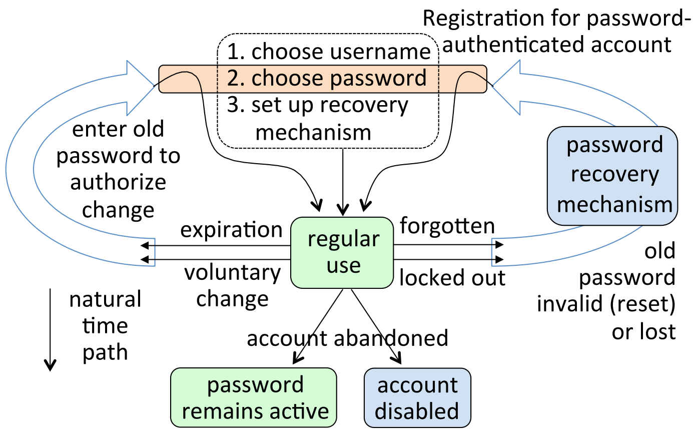

Registration for password-authenticated account
1. choose username
2. choose password
3. set up recovery mechanism
enter old password to authorize change
password recovery mechanism
expiration
regular use
forgotten
voluntary change
locked out
old password invalid (reset) or lost
natural time path
account abandoned
password remains active
account disabled

Figure 1.7: Password-authenticated account lifecycle. Lifecycle diagrams help in threat modeling. When primary user authentication involves biometrics (Chapter 3), a fallback mechanism is also typically required, presenting additional attack surface for analysis.

media, or cloud storage?) Revisiting your diagram, add in the locations of all authorized users, and the communications paths they are expected to use. (Your diagram is becoming a bit crowded, you say? Redraw pictures as necessary.) Are any paths missing—how about users logging in by VPN from home offices? Are all communications links shown, both wireline and wireless? Might an authorized remote user gain access through a Wi-Fi link in a café, hotel or airport—could that result in a middle-person scenario (Chapter 4), with data in the clear, if someone nearby has configured a laptop as a rogue wireless access point that accepts and then relays communications, serving as a proxy to the expected access point? Might attackers access or alter data that is sent over any of these links?

Revisit your diagram again. (Is this sounding familiar?) Who installs new hardware, or maintains hardware? Do consultants or custodial staff have intermittent access to offices? The diagram is just a starting point, to focus attention on something concrete. Suggestions serve to cause the diagram to be looked at in different ways, expanded, or refined to lower levels of detail. The objective is to encourage semi-structured brainstorming, get a stream of questions flowing, and stimulate free thought about possible threats and attack vectors—beyond staring at a blank page. So begins threat modeling, an open-ended task.

1.5.2 Attack trees for threat modeling

Attack trees are another useful threat modeling tool, especially to identify attack vectors. A tree starts with a *root node* at the top, labeled with an overall attack goal (e.g., enter a house). Lower nodes break out alternative ways to reach their parent's goal (e.g., enter through a window, through a door, tunnel into the basement). Each may similarly be broken down further (e.g., open an unlocked window, break a locked window). Each internal node is the root of a subtree whose children specify ways of reaching it. Subtrees end in *leaf nodes*. A path connecting a leaf node to the root lists the steps (attack vector) composing one full attack; intermediate nodes may detail prerequisite steps, or classify

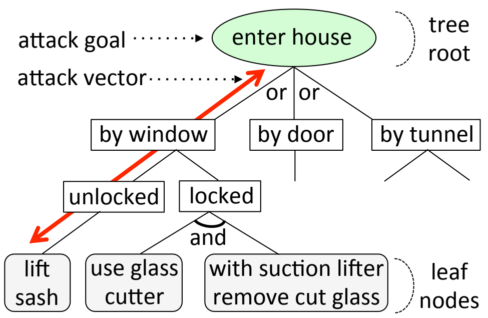


Figure 1.8: Attack tree. An attack vector is a full path from root to leaf.

different vectors. Multiple children of a node (Fig. 1.8) are by default distinct alternatives (logical OR nodes); however, a subset of nodes at a given level can be marked as an AND set, indicating that all are jointly necessary to meet the parent goal. Nodes can be annotated with various details—e.g., indicating a step is infeasible, or by values indicating costs or other measures. The attack information captured can be further organized, often suggesting natural classifications of attack vectors into known categories of attacks.

The main output is an extensive (but usually incomplete) list of possible attacks, e.g., Figure 1.9. The attack paths can be examined to determine which ones pose a risk in the real system; if the circumstances detailed by a node are for some reason infeasible in the target system, the path is marked invalid. This helps maintain focus on the most relevant threats. Notice the asymmetry: an attacker need only find one way to break into a system, while the defender (security architect) must defend against all viable attacks.

An attack tree can help in forming a security policy, and in security analysis to check that mechanisms are in place to counter all identified attack vectors, or explain why particular vectors are infeasible for relevant adversaries of the target system. Attack vectors identified may help determine the types of defensive measures needed to protect specific assets from particular types of malicious actions. Attack trees can be used to prioritize vectors as high or low, e.g., based on their ease, and relevant classes of adversary.

The attack tree methodology encourages a form of directed brainstorming, adding structure to what is otherwise an ad hoc task. The process benefits from a creative mind. It requires a skill that improves with experience. The process is also best used iteratively, with a tree extended as new attacks are identified on review by colleagues, or merged with trees independently constructed by others. Attack trees motivate security architects to "think like attackers", to better defend against them.

Example *(Enumerating password authentication attacks).* To construct a list of attacks on password authentication, one might draw a data flow diagram showing a password's end-to-end paths, then identify points where an attacker might try to extract information. An alternative approach is to build an attack tree, with root goal to gain access to a user's account on a given system. Which method is used is a side detail towards the desired output: a list of potential attacks to consider further (Figure 1.9).

‡**Exercise** (Free-lunch attack tree). Read the article by Mauw [12], for a fun example of an attack tree. As supplementary reading see *attack-defense trees* by Kordy [10].

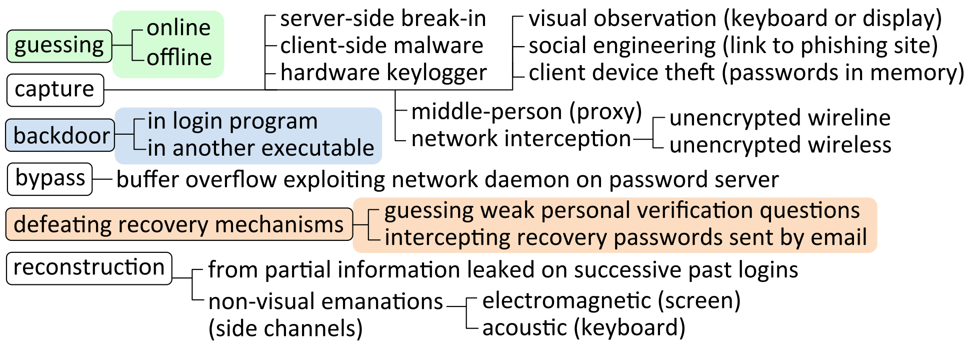

guessing
online
offline
capture
server-side break-in
client-side malware
hardware keylogger
visual observation (keyboard or display)
social engineering (link to phishing site)
client device theft (passwords in memory)
backdoor
in login program
in another executable
middle-person (proxy)
network interception
unencrypted wireline
unencrypted wireless
bypass
buffer overflow exploiting network daemon on password server
defeating recovery mechanisms
guessing weak personal verification questions
intercepting recovery passwords sent by email
reconstruction
from partial information leaked on successive past logins
non-visual emanations (side channels)
electromagnetic (screen)
acoustic (keyboard)

Figure 1.9: Attacks on password-based authentication. Such a list may be created by diagram-driven approaches, attack trees, or other methods. Terms and techniques in this chart are explained in later chapters, including discussion of passwords in Chapter 3.

‡**Exercise** (Recovering screen content). Build an attack tree with goal to extract data shown on a target device display. Consider two cases: desktop and smartphone screens.

1.5.3 Other threat modeling approaches: checklists and STRIDE

ATTACK/THREAT CHECKLISTS. While diagram-driven threat modeling and attack trees are fairly free-form, and at best semi-structured, at the other end of the spectrum is the idea of consulting fixed attack checklists, drawn up over time from past experience by larger communities, and accompanied by varying levels of supporting detail.

Advantages of this are that extensive such checklists exist; their thorough nature can help ensure that well-known threats are not overlooked by ad hoc processes; and compared to previously mentioned approaches, they may require less experience or provide better learning opportunities. Disadvantages are that such pre-constructed generic lists contain known attacks in generalized terms, without taking into account unique details and assumptions of the target system and environment in question—they may thus themselves overlook threats relevant to particular environments and designs; long checklists risk becoming tedious, replacing a security analyst's creativity with boredom; and their length may distract attention away from higher-priority threats. Checklists are perhaps best used as a complementary tool, or in a hybrid method as a cross-reference when pursuing a diagram-driven approach.

STRIDE. Another approach uses a small set of keywords to stimulate thought, unburdened by a longer list. A specific such method is *STRIDE* (Section 1.9 gives citations). This acronym is a memory aid for recalling the following six categories of threats:

- Spoofing—attempts to impersonate a thing (e.g., web site), or an entity (e.g., user).
- Tampering—unauthorized altering, e.g., of code, stored data, transmitted packets.
- Repudiation—denying responsibility for past actions.
- Information disclosure—unauthorized release of data.
- Denial of service—impacting availability of services, or the quality of services, through malicious actions that consume resources or induce errors in systems.

- Escalation of privilege—obtaining privileges to access resources, typically referring to malware that gains a base level of access as a foothold and then exploits vulnerabilities to extend this to gain greater access.

While these six categories are not definitive or magical, they are useful in that most threats can be associated with one of these categories. The idea is to augment the diagram-driven approach by considering, at each point where the question "Where can things break?" is asked, whether any of these six categories of problems might occur. STRIDE thus offers another way to stimulate open-ended thought while looking at a diagram and trying to identify threats to architectural components—in this case, guided by six keywords.

1.6 Model-reality gaps and real-world outcomes

We consider why threat modeling is difficult, before returning to check that what we have discussed helps deliver on the goals set out in defining security policies.

1.6.1 Threat modeling and model-reality gaps

Threat modeling is tricky. An example illustrates the challenge of anticipating threats.

Example *(Hotel safebox).* You check into a hotel in a foreign country. You do not speak the language well. The hotel staff appears courteous, but each time you visit the lobby it seems a different face is behind the front desk. Your room has a small safe, the electronic type with its door initially open (unlocked), allowing guests at that point to choose their own combination. Upon three incorrect attempted combinations, the safe enters a lockout mode requiring hotel maintenance to reset it. You choose a combination, remember it, stash your money, close the box, then go swimming. Is your money secure? The natural assumption is that a thief, trying to open the lock by guessing your combination, will with high probability guess wrong three times, leaving your money safe inside. But, do you trust the hotel staff? Someone in maintenance must know the reset or master combination to allow for hotel guests who either forget their combination or enter three wrong tries. If you return and the money is gone, do you trust the hotel maintenance staff to help you investigate? Can you trust the hotel management? And the local police—are they related to hotel staff or the owner? (What implicit assumptions have you made?)

QUALITY OF A THREAT MODEL. A threat model's quality, with respect to protecting a particular system, depends on how accurately the model reflects details of that system and its operating environment. A mismatch between model and reality can give a dangerously false sense of security. Some model-reality gaps arise due to the abstraction process inherent in modeling—it is difficult for a high-level, abstract model to encapsulate all technical details of a system, and *details are important in security*. Major gaps often arise from two related modeling errors:

1. invalid assumptions (often including misplaced trust); and
2. focus on the wrong threats.

Both can result from failing to adapt to changes in technology and attack capabilities. Model assumptions can also be wrong, i.e., fail to accurately represent a target system, due to incomplete or incorrect information, over-simplification, or loss of important details through abstraction. Another issue is failure to record assumptions explicitly—implicit assumptions are rarely scrutinized. Focusing attention on the wrong threats may mean wasting effort on threats of lower probability or impact than others. This can result not only from unrealistic assumptions but also from: inexperience or lack of knowledge, failure to consider all possible threats (incompleteness), new vulnerabilities due to computer system or network changes, or novel attacks. It is easy to instruct someone to defend against all possible threats; anticipating the unanticipated is more difficult.

WHAT'S YOUR THREAT MODEL. Ideally, threat models are built using both practical experience and analytical reasoning, and continually adapted to inventive attackers who exploit rapidly evolving software systems and technology. Perhaps the most important security analysis question to always ask is: *What's your threat model?* Getting the threat model wrong—or getting only part of it right—allows many successful attacks in the real world, despite significant defensive expenditures. We give a few more examples.

Example *(Online trading fraud).* A security engineer models attacks on an online stock trading account. To stop an attacker from extracting money, she disables the ability to directly remove cash from such accounts, and to transfer funds across accounts. The following attack nonetheless succeeds. An attacker X breaks into a victim account by obtaining its userid and password, and uses funds therein to bid up the price of a thinly traded stock, which X has previously purchased at lower cost on his own account. Then X sells his own shares of this stock, at this higher price. The victim account ends holding the higher-priced shares, bought on the (manipulated) open market.

Example *(Phishing one-time passwords).* Some early online banks used *one-time passwords*, sharing with each account holder a sheet containing a list of unique passwords to be used once each from top to bottom, and crossed off upon use—to prevent repeated use of passwords stolen (e.g., by phishing or malicious software). Such schemes have nonetheless been defeated by tricking users to visit a fraudulent version of a bank web site, and requesting entry of the next five listed passwords "to help resolve a system problem". The passwords entered are used once each on the real bank site, by the attacker. (Chapter 3 discusses one-time passwords and the related mechanism of *passcode generators*.)

Example *(Bypassing perimeter defenses).* In many enterprise environments, corporate gateways and firewalls selectively block incoming traffic to protect local networks from the external Internet. This provides no protection from employees who, bypassing such *perimeter defenses*, locally install software on their computers, or directly connect by USB port memory tokens or smartphones for synchronization. A well-known attack vector exploiting this is to sprinkle USB tokens (containing malicious software) in the parking lot of a target company. Curious employees facilitate the rest of the attack.

DEBRIEFING. What went wrong in the above examples? The assumptions, the threat model, or both, were incorrect. Invalid assumptions or a failure to accurately model the operational environment can undermine what appears to be a solid model, despite convincing security arguments and mathematical proofs. One common trap is failing to validate assumptions: if a security proof relies on assumption A (e.g., hotel staff are honest), then the logical correctness of the proof (no matter how elegant!) does not provide protection if in the current hotel, A is false. A second is that a security model may truly provide a 100% guarantee that all attacks it considers are precluded by a given defense, while in practice the modeled system is vulnerable to attacks that the model fails to consider.

ITERATIVE PROCESS: EVOLVING THREAT MODELS. As much art as science, threat modeling is an iterative process, requiring continual adaptation to more complete knowledge, new threats and changing conditions. As environments change, static threat models become obsolete, failing to accurately reflect reality. For example, many Internet security protocols are based on the original *Internet threat model*, which has two core assumptions: (1) endpoints, e.g., client and server machine, are trustworthy; and (2) the communications link is under attacker control (e.g., subject to eavesdropping, message modification, message injection). This follows the historical cryptographer's model for securing data transmitted over unsecured channels in a hostile communications environment. However, assumption (1) often fails in today's Internet where malware (Chapter 7) has compromised large numbers of endpoint machines.

Example *(Hard and soft keyloggers).* Encrypting data between a client machine and server does not protect against malicious software that intercepts keyboard input, and relays it to other machines electronically. The hardware variation is a small, inexpensive memory device plugged in between a keyboard cable and a computer, easily installed and removed by anyone with occasional brief office access, such as cleaning staff.

1.6.2 Tying security policy back to real outcomes and security analysis

Returning to the big picture, we now pause to consider: How does "security" get tied back to "security policy", and how does this relate to threat models and security mechanisms?

OUTCOME SCENARIOS. Security defenses and *mechanisms* (means to implement defenses) are designed and used to support security policies and services as in Fig. 1.1. Consider the following outcomes relating defenses to security policies.

1. The defenses fail to properly support the policy; the security goal is not met.
2. The defenses succeed in preventing policy violations, and the policy is complete in the sense of fully capturing an organization's security requirements. The resulting system is "secure" (both relative to the formal policy and real-world expectations).
3. The formal policy does not fully capture actual security requirements. Here, even if defenses properly support policy (attaining "security" relative to the formal policy), the real-world common-sense expectation of security might not be met.

The third case motivates the following advice: *Whenever ambiguous words like "secure" and "security" are used, request that their intended meaning and context be clarified.*

SECURITY ANALYSIS AND KEY QUESTIONS. Figure 1.10 provides overall context for the iterative process of security design and analysis. It may proceed as follows. Identify the valuable assets. Determine suitable forms of protection to counter identified threats and vulnerabilities; adversary modeling and threat modeling help here. This helps

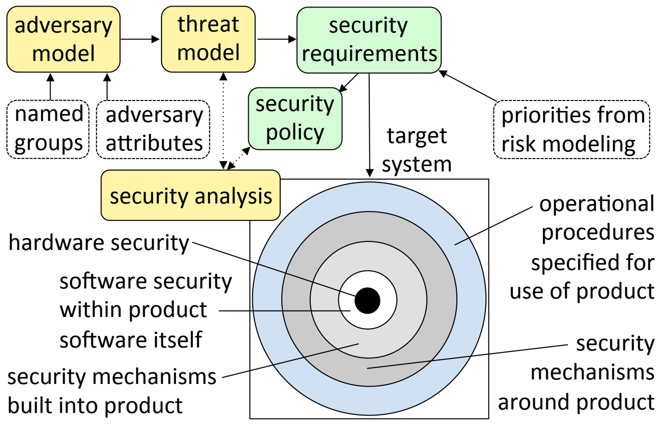

adversary model
threat model
security requirements
named groups
adversary attributes
security policy
target system
priorities from risk modeling
security analysis
hardware security
software security within product
software itself
security mechanisms built into product
operational procedures specified for use of product
security mechanisms around product

Figure 1.10: Security analysis in context. Security analysis is relative to a threat model defining in-scope attacks, and a security policy implying protection goals and authorized actions. Security analysis aims to confirm that defenses in place address the identified threats, and explains how. Compare to Figure 1.4's software design lifecycle.

refine security requirements, shaping the security policy, which in turn shapes system design. Security mechanisms that can support the policy in the target environment are then selected. As always, key questions help:

- What assets are valuable? (Alternatively: what are your protection goals?)
- What potential attacks put them at risk?
- How can potentially damaging actions be stopped or otherwise managed?

Options to mitigate future damage include not only attack *prevention* by countermeasures that preclude (or reduce the likelihood of) attacks successfully exploiting vulnerabilities, but also *detection*, *real-time response*, and *recovery* after the fact. Quick recovery can reduce impact. Consequences can also be reduced by insurance (Section 1.3).

TESTING IS NECESSARILY INCOMPLETE. Once a system is designed and implemented, how do we test that the protection measures work and that the system is "secure"? (Here you should be asking: What definition of "secure" are you using?) How to test whether security requirements have been met remains without a satisfactory answer. Section 1.4 mentioned security analysis (often finding design flaws), third-party security evaluation, and pen testing (often finding implementation and configuration flaws). Using checklist ideas from threat modeling, testing can be done based on large collections of common flaws, as a form of security-specific regression testing; specific, known attacks can be compiled and attempted under controlled conditions, to see whether a system successfully withstands them. This of course leaves unaddressed attacks not yet foreseen or invented, and thus difficult to include in tests. Testing is also possible only for certain classes of attacks. Assurance is thus incomplete, and often limited to well-defined scopes.

SECURITY IS UNOBSERVABLE. In regular software engineering, verification involves testing specific features for the presence of correct outcomes given particular inputs. In contrast, security testing would ideally also confirm the *absence* of exploitable flaws. This may be called a *negative goal*, among other types of *non-functional goals*. To repeat: we want not only to verify that expected functionality works as planned, but also that exploitable artifacts are absent. This is not generally possible—aside from the difficulty of proving properties of software at scale, the universe of potential exploits is unknown. Traditional functional and feature testing cannot show the absence of problems; this distinguishes security. Security guarantees may also evaporate due to a small detail of one component being updated or reconfigured. A system's security properties are thus difficult to predict, measure, or see; we cannot observe security itself or demonstrate it, albeit on observing undesirable outcomes we know it is missing. Sadly, *not* observing bad outcomes does not imply security either—bad things that are unobservable could be latent, or be occurring unnoticed. The security of a computer system is not a testable feature, but rather is said (unhelpfully) to be *emergent*—resulting from the complex interaction of elements that compose an entire system.

ASSURANCE IS DIFFICULT, PARTIAL. So then, what happens in practice? Evaluation criteria are altered by experience, and even thorough security testing cannot provide 100% guarantees. In the end, we seek to iteratively improve security policies, and likewise confidence that protections in place meet security policy and/or requirements. *Assurance* of this results from sound design practices, testing for common flaws and known attacks using available tools, formal modeling of components where suitable, ad hoc analysis, and heavy reliance on experience. The best lessons often come from attacks and mistakes.

1.7 ‡Design principles for computer security

No complete checklist exists—neither short nor long—that system designers can follow to guarantee that computer-based systems are "secure". The reasons are many, including large variations across technologies, environments, applications and requirements. Section 1.5.3 discusses partial checklists, but independently, system designers are encouraged to understand and follow a set of widely applicable security design principles. We collect them here together in one place, and revisit them throughout the book with in-context examples to aid understanding.

P1 SIMPLICITY-AND-NECESSITY: Keep designs as simple and small as possible. Reduce the number of components used, retaining only those that are essential; minimize functionality, favor minimal installs, and disable unused functionality. Economy and frugality in design simplifies analysis and reduces errors and oversights. Configure initial deployments to have non-essential services and applications disabled by default (related to P2).

NOTE. This principle among others supports another broad principle: *minimizing attack surface*. Every interface that accepts external input or exposes programmatic functionality provides an entry point for an attacker to change or acquire a program control path (e.g., install code or inject commands for execution), or alter data that

‡Design principles used throughout the book are collected in this section for unified reference. Readers may prefer to skip this section and refer back for relevant principles as they arise later.

might do likewise. The goal is to minimize the number of interfaces, simplify their design (to reduce the number of ways they might be abused), minimize external access to them, and restrict such access to authorized parties. Importantly, security mechanisms themselves should not introduce new exploitable attack surfaces.

P2 SAFE-DEFAULTS: Use safe default settings; remember defaults often go unchanged. For access control, deny-by-default. Favor explicit inclusion over exclusion—use *whitelists*, listing authorized parties (all others being denied), rather than *blacklists* of parties to be denied access (all others allowed). Design services to be *fail-safe*, meaning that they fail "closed" (denying access) rather than "open".

NOTE. A related idea, e.g., for data sent over real-time links, is to encrypt by default using *opportunistic encryption*—encrypting session data whenever supported by the receiving end. In contrast, default encryption is not generally recommended in all cases for *stored data*, as the importance of confidentiality must be weighed against the complexity of long-term key management and the risk of permanent loss of data if encryption keys are lost; for session data, immediate decryption upon receipt at the endpoint recovers cleartext.

P3 OPEN-DESIGN: Do not rely on secret designs, attacker ignorance, or *security by obscurity*. Invite and encourage open review and analysis. Example: undisclosed cryptographic algorithms are now widely discouraged—the *Advanced Encryption Standard* was selected from a set of public candidates by open review. Without contradicting this, leverage unpredictability where advantageous, as arbitrarily publicizing tactical defense details is rarely beneficial (there is no gain in advertising to thieves that you are on vacation, or posting house blueprints). Be reluctant to leak secret-dependent error messages or timing data, lest they be useful to attackers.

NOTE. This principle is related to *Kerckhoffs' principle*—a system's security should not rely upon the secrecy of its design details.

P4 COMPLETE-MEDIATION: For each access to every object, and ideally immediately before the access is to be granted, verify proper authority. Verifying authorization requires authentication (corroboration of an identity), checking that the associated principal is authorized, and checking that the request has integrity (it must not be modified after being issued by the legitimate party—cf. P19).

P5 ISOLATED-COMPARTMENTS: Compartmentalize system components using strong isolation structures that prevent cross-component communication or leakage of information and control. This limits damage when failures occur, and protects against *escalation of privileges* (Chapter 6); P6 and P7 have similar motivations. Restrict authorized cross-component communication to observable paths with defined interfaces to aid mediation, screening, and use of *chokepoints*. Examples of containment means include: process and memory isolation, disk partitions, virtualization, software guards, zones, gateways and firewalls.

NOTE. *Sandbox* is a term used for mechanisms offering some form of isolation.

P6 LEAST-PRIVILEGE: Allocate the fewest privileges needed for a task, and for the shortest duration necessary. For example, retain superuser privileges (Chapter 5) only for actions requiring them; drop and reacquire privileges if needed later. Do not use a Unix *root* account for tasks doable with regular user privileges. This reduces exposure, and limits damage from the unexpected. P6 complements P5 and P7.

NOTE. This principle is related to the military *need-to-know* principle—access to sensitive information is granted only if essential to carrying out one's official duties.

P7 MODULAR-DESIGN: Avoid designing monolithic modules that concentrate large privilege sets in single entities; favor object-oriented and finer-grained designs (e.g., Linux *capabilities*) segregating privileges across smaller units, multiple processes or distinct principals. P6 provides guidance where monolithic designs already exist.

NOTE. This principle is related to the financial accounting principle of *separation of duties*—related duties are assigned to independent parties so that an *insider attack* requires collusion. This also differs from requiring *multiple authorizations* from distinct parties (e.g., two keys or signatures to open a safety-deposit box or authorize large-denomination cheques), a generalization of which is *thresholding* of privileges—requiring k of t parties ($2 \leq k \leq t$) to authorize an action.

P8 SMALL-TRUSTED-BASES: Strive for small code size for components that must be trusted, i.e., components on which a larger system strongly depends for security. Example 1: high-assurance systems centralize critical security services in a minimal core operating system *microkernel* (cf. Chapter 5 end notes), whose smaller size allows efficient concentration of security analysis. Example 2: cryptographic algorithms separate mechanisms from secrets, with trust requirements reduced to a *secret key* changeable at far less cost than the cryptographic algorithm itself.

NOTE. This principle is related to the *minimize-secrets* principle—secrets should be few in number. One motivation is to reduce management complexity.

P9 TIME-TESTED-TOOLS: Rely wherever possible on time-tested, expert-built security tools including protocols, cryptographic primitives and toolkits, rather than designing and implementing your own. History shows that security design and implementation is difficult to get right even for experts; thus amateurs are heavily discouraged (*don't reinvent a weaker wheel*). Confidence increases with the length of time mechanisms and tools have survived (sometimes called *soak testing*).

NOTE. This principle's underlying reasoning is that a widely used, heavily scrutinized mechanism is less likely to retain flaws than many independent, scantly reviewed implementations. Thus using crypto libraries like OpenSSL, that are well-known, is encouraged. Less understood is an older *least common mechanism* principle: minimize the number of mechanisms (shared variables, files, system utilities) shared by two or more programs and depended on by all. It recognizes that interdependencies increase risk. *Code diversity* can also reduce impacts of single flaws.

P10 LEAST-SURPRISE: Design mechanisms, and their user interfaces, to behave as users expect. Align designs with users' mental models of their protection goals, to reduce user mistakes. Especially where errors are irreversible (e.g., sending confidential data or secrets to outside parties), tailor to the experience of target users; beware designs suited to trained experts but unintuitive or triggering mistakes by ordinary users. Simpler, easier-to-use (i.e., *usable*) mechanisms yield fewer surprises.

P11 USER-BUY-IN: Design security mechanisms that users are motivated to use, to promote regular cooperative use; and so that users' path of least resistance is a safe path. Seek design choices that illuminate benefits, improve user experience, and minimize inconvenience. Mechanisms viewed as time-consuming, inconvenient or without perceived benefit encourage bypassing and non-compliance. Example: a subset of Google gmail users voluntarily use a two-step authentication scheme, which augments basic passwords by one-time passcodes sent to the user's phone.

P12 SUFFICIENT-WORK-FACTOR: For mechanisms susceptible to direct *work-factor* calculation, design the security mechanism so that the work cost to defeat it clearly exceeds the resources of expected attackers. Use defenses suitably strong to protect against anticipated classes of adversaries. Example 1: random cryptographic keys must be long enough that they cannot be found by brute-force search. Example 2: user-chosen passwords should be disallowed if they are so weak that a small number of guesses yields a non-negligible chance of success.

P13 DEFENSE-IN-DEPTH: Build defenses in multiple layers backing each other up, forcing attackers to defeat independent layers. Avoid *single points of failure*. If an individual layer relies on several defense segments, design each to be comparably strong ("equal-height fences") and strengthen the weakest segment first (smart attackers jump the lowest bar or break the *weakest link*). As a design assumption, assume some defenses will fail on their own due to errors, and that attackers will defeat others more easily than expected or entirely bypass them.

P14 EVIDENCE-PRODUCTION: Record system activities through event logs and by other means to promote accountability, help understand and recover from system failures, and support intrusion detection tools. Example: robust audit trails complement *forensic analysis* tools, to help reconstruct events related to intrusions and criminal activities. In many cases, mechanisms that facilitate attack detection and evidence production may be more cost-effective than outright prevention.

P15 DATA-TYPE-VERIFICATION: Verify that all received data conforms to expected or assumed properties. If data input is expected, ensure that it cannot be processed as code by subsequent components. Example: this may be part of *input sanitization* and *canonicalization* (e.g., of fragmented packets or encoded characters in URLs) to address *code injection* and *command injection* attacks (Chapter 9).

P16 REMNANT-REMOVAL: On termination of a session or program, remove all traces of sensitive data associated with a task, including secret keys and any remnants recoverable from secondary storage, RAM and cache memory. Note that common file deletion removes directory entries, whereas *secure deletion* aims to make file content unrecoverable even by forensic tools. Related to remnant removal, beware that while a process is active, information may leak elsewhere by *side channels*.

P17 TRUST-ANCHOR-JUSTIFICATION: Ensure or justify confidence placed in any base point of assumed trust, especially when mechanisms iteratively or transitively extend trust from a base point (such as a *trust anchor* in a browser *certificate chain*, Chapter 8). More generally, verify trust assumptions where possible, with extra diligence at registration, initialization, software installation, and starting points in the lifecycle of a software application, security key or credential.

P18 INDEPENDENT-CONFIRMATION: Use simple, independent cross-checks to increase confidence in code or data, especially when it is potentially provided by outside domains or over untrusted channels. Example: integrity of downloaded software applications or public keys can be confirmed (Chapter 8) by comparing a locally computed *cryptographic hash* (Chapter 2) of the item to a "known-good" hash obtained over an independent channel (voice call, text message, widely trusted site).

P19 REQUEST-RESPONSE-INTEGRITY: Verify that responses match requests in *name resolution* protocols and other distributed protocols. Their design should detect message alteration or substitution, and include cryptographic integrity checks that bind steps to each other within a given transaction or protocol run; beware protocols lacking authentication. Example: a *certificate request* specifying a unique subject name or domain expects in response a certificate for that subject; this field in the response certificate should be cross-checked against the request.

P20 RELUCTANT-ALLOCATION: Be reluctant to allocate resources or expend effort in interactions with unauthenticated, external agents. For processes or services with special privileges, be reluctant to act as a conduit extending such privileges to unauthenticated (untrusted) agents. Place a higher burden of proof of identity or authority on agents that initiate a communication or interaction. (A party initiating a phone call should not be the one to demand: *Who are you?*) Failure to follow this principle facilitates various *denial of service attacks* (Chapter 11).

NOTE. Reluctance also arises in P3, in terms of leaking data related to secrets.

We also include two higher-level principles and a maxim.

HP1 SECURITY-BY-DESIGN: Build security in, starting at the initial design stage of a development cycle, since secure design often requires core architectural support absent if security is a late-stage add-on. Explicitly state the *design goals* of security mechanisms and what they are *not* designed to do, since it is impossible to evaluate effectiveness without knowing goals. In design and analysis documents, explicitly state all security-related *assumptions*, especially related to trust and trusted parties (supporting P17); note that a security policy itself might not specify assumptions.

HP2 DESIGN-FOR-EVOLUTION: Be mindful of evolution when designing base architectures, mechanisms, and protocols. Example: design systems with *algorithm agility*, so that upgrading a crypto algorithm (e.g., encryption, hashing) is graceful and does not impact other system components. A related management process is to regularly re-evaluate the effectiveness of security mechanisms, in light of evolving threats, technology, and architectures—being ready to update designs as needed.

VERIFY FIRST. The diplomatic maxim "*trust but verify*" suggests that given assertions by foreign diplomats whom you don't actually trust, one should feign trust while silently cross-checking for yourself. In computer security, the rule is better stated as: *verify first (before trusting)*. Design principles related to this idea include: COMPLETE-MEDIATION (P4), DATA-TYPE-VERIFICATION (P15), TRUST-ANCHOR-JUSTIFICATION (P17), and INDEPENDENT-CONFIRMATION (P18).

1.8 ‡Why computer security is hard

Many of today's fundamental problems in computer security remain from decades ago, despite huge changes in computing hardware, software, applications and environments. For example, the first large-scale Internet security incident attracting widespread public attention was the 1988 *Internet worm* (Chapter 7). *Code Red* and related computer worms of 2001-2003 used remarkably similar vectors to spread, but had larger impact due to the growth in the Internet user population. This highlights both the value of learning general computer security principles—solid principles remain true over time—and the difficulty in applying these to improve real-world security, due to various challenging aspects of computer security. We list a number of these aspects here, for context and discussion:

1. *intelligent, adaptive adversary*: while most science relies on nature not being capricious, computer security faces an intelligent, active adversary who learns and adapts, and is often economically motivated.
2. *no rulebook*: attackers are not bound to any rules of play, while defenders typically follow protocol conventions, interface specifications, standards and customs.
3. *defender-attacker asymmetry*: attackers need find only one weak link to exploit, while defenders must defend all possible attack points.
4. *scale of attack*: the Internet enables attacks of great scale at little cost—electronic communications are easily reproduced and amplified, with increasing bandwidth and computing power over time.
5. *universal connectivity*: growing numbers of Internet devices with any-to-any packet transmission abet geographically distant attackers (via low traceability/physical risk).
6. *pace of technology evolution*: rapid technical innovation means continuous churn in hardware devices and software systems, continuous software upgrades and patches.
7. *software complexity*: the size and complexity of modern software platforms continuously grows, as does a vast universe of application software. Software flaws may also grow in number more than linearly with number of lines of code.
8. *developer training and tools*: many software developers have little or no security training; automated tools to improve software security are difficult to build and use.
9. *interoperability* and *backwards compatibility*: interoperability requirements across diverse hardware-software and legacy systems delays and complicates deploying security upgrades, resulting in ongoing vulnerabilities even if updates are available.

‡The remainder of this book helps us understand many of the aspects listed in this section. While given here, we encourage readers to return to this list as a summary, on completing the final chapter.

10. *market economics and stakeholders*: market forces often hinder allocations that improve security, e.g., stakeholders in a position to improve security, or who would bear the cost of deploying improvements, may not be those who would gain benefit.
11. *features beat security*: while it is well accepted that complexity is the enemy of security (cf. P1), little market exists for simpler products with reduced functionality.
12. *low cost beats quality*: low-cost low-security wins in "*market for lemons*" scenarios where to buyers, high-quality software is indistinguishable from low (other than costing more); and when software sold has no liability for consequential damages.
13. *missing context of danger and losses*: cyberspace lacks real-world context cues and danger signals to guide user behavior, and consequences of security breaches are often not immediately visible nor linkable to the cause (i.e., the breach itself).
14. *managing secrets is difficult*: core security mechanisms often rely on secrets (e.g., crypto keys and passwords), whose proper management is notoriously difficult and costly, due to the nature of software systems and human factors.
15. *user non-compliance (human factors)*: users bypass or undermine computer security mechanisms that impose inconveniences without visible direct benefits (in contrast: physical door locks are also inconvenient, but benefits are understood).
16. *error-inducing design (human factors)*: it is hard to design security mechanisms whose interfaces are intuitive to learn, distinguishable from interfaces presented by attackers, induce the desired human actions, and resist *social engineering*.
17. *non-expert users (human factors)*: whereas users of early computers were technical experts or given specialized training under enterprise policies, today many are non-experts without formal training or any technical computer background.
18. *security not designed in*: security was not an original design goal of the Internet or computers in general, and retro-fitting it as an add-on feature is costly and often impossible without major redesign (see principle HP1).
19. *introducing new exposures*: the deployment of a protection mechanism may itself introduce new vulnerabilities or attack vectors.
20. *government obstacles*: government desire for access to data and communications (e.g., to monitor criminals, or spy on citizens and other countries), and resulting policies, hinders sound protection practices such as strong encryption by default.

We end by noting that this is but a partial list! Rather than being depressed by this, as optimists we see a great opportunity—in the many difficulties that complicate computer security, and in technology trends suggesting challenges ahead as critical dependence on the Internet and its underlying software deepens. Both emphasize the importance of understanding what can go wrong when we combine people, computing and communications devices, and software-hardware systems.

We use computers and mobile devices every day to work, communicate, gather information, make purchases, and plan travel. Our cars rely on software systems—as do our airplanes. (Does this worry you? What if the software is wirelessly updated, and the source of updates is not properly authenticated?) The business world comes to a standstill when Internet service is disrupted. Our critical infrastructure, from power plants and electricity grids to water supply and financial systems, is dependent on computer hardware, software and the Internet. Implicitly we expect, and need, security and dependability.

Perhaps the strongest motivation for individual students to learn computer security (and for parents and friends to encourage them to do so) is this: security expertise may be today's very best job-for-life ticket, as well as tomorrow's. It is highly unlikely that software and the Internet itself will disappear, and just as unlikely that computer security problems will disappear. But beyond employment for a lucky subset of the population, having a more reliable, trustworthy Internet is in the best interest of society as a whole. The more we understand about the security of computers and the Internet, the safer we can make them, and thereby contribute to a better world.

1.9 ‡End notes and further reading

Many *formal security models*, including for specific policies related to confidentiality, integrity, and access control, are discussed in the research literature and older books—see for example Pfleeger [15]. Gollmann [7, Chapter 13] discusses *formal security evaluation*. Rescorla [16, page 1] mentions the Internet threat model. Attack trees are related to *threat trees* [2] and predated by 1960s-era *fault tree analysis*. Related Kuang decision trees arising from Baldwin [5] influenced later work on vulnerability assessment (cf. [4, page 143]). MITRE maintains a Common Attack Pattern Enumeration and Classification (CAPEC), an extensive list (dictionary and classification) of security attacks that aims to aid defenders; such collections, often with supporting information including prototype attack code, may be called *attack libraries*. Shostack [19] gives an authoritative treatment of threat modeling (including STRIDE); see also Howard [8]. Lowry [11] discusses adversary modeling. Akerlof [1] explains the "market for lemons" and what happens when buyers cannot distinguish low-quality products from (more costly) high-quality products.

Additional barriers related to risk assessment and *security metrics* are noted by Jaquith [9, pp. 31-36] and Parker [14]. As examples, risk assessment equations can be highly sensitive to changes in often arbitrary modeling assumptions; outliers can dominate analysis when modeling rare, high-impact events; and risk estimates are complicated by software complexity and human factors issues. For qualitative *risk assessment*, see risk assessment standards and guidelines (Table 1.1 is based on NIST SP 800-30 [13]); textbooks giving examples include Basin [6, §8.4] and Stallings [21, §14.5]. Core *security design principles* from 1975 by Saltzer and Schroeder [18] have been periodically revisited, for example, by Saltzer and Kaashoek [17, Ch. 11] and Smith [20]; see also Basin [6] for examples related to long-standing principles.

Our definition of *attack* herein is somewhat narrow. For broader discussion of relationships between *dependability*, security and *trustworthiness*, and definitions of *faults* (including malicious faults), *failures* and *errors*, see Avizienis [3].

‡*The double-dagger symbol denotes sections that may be skipped on first reading, or by instructors using the book for time-constrained courses.*

References

[1] G. A. Akerlof. The market for “lemons”: Quality uncertainty and the market mechanism. *The Quarterly Journal of Economics*, 84(3):488–500, August 1970.

[2] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994. Includes author’s list of 25 Greatest Works in Computer Security.

[3] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *ACM Trans. Inf. Systems and Security*, 1(1):11–33, 2004.

[4] R. G. Bace. *Intrusion Detection*. Macmillan, 2000.

[5] R. W. Baldwin. *Rule Based Analysis of Computer Security*. Ph.D. thesis, MIT, Cambridge, MA, June 1987. Describes security checkers called Kuang systems, and in particular one built for Unix.

[6] D. Basin, P. Schiller, and M. Schläpfer. *Applied Information Security*. Springer, 2011.

[7] D. Gollmann. *Computer Security (3rd edition)*. John Wiley, 2011.

[8] M. Howard and D. LeBlanc. *Writing Secure Code (2nd edition)*. Microsoft Press, 2002.

[9] A. Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Addison-Wesley, 2007.

[10] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer. Foundations of attack-defense trees. In *Formal Aspects in Security and Trust 2010)*, pages 80–95. Springer LNCS 6561 (2011).

[11] J. Lowry, R. Valdez, and B. Wood. Adversary modeling to develop forensic observables. In *Digital Forensics Research Workshop (DFRWS)*, 2004.

[12] S. Mauw and M. Oostdijk. Foundations of attack trees. In *Information Security and Cryptology (ICISC 2005)*, pages 186–198. Springer LNCS 3935 (2006).

[13] NIST. Special Pub 800-30 rev 1: Guide for Conducting Risk Assessments. U.S. Dept. of Commerce, September 2012.

[14] D. B. Parker. Risks of risk-based security. *Comm. ACM*, 50(3):120–120, March 2007.

[15] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing (4th edition)*. Prentice Hall, 2006.

[16] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[17] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, 2010.

[18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[19] A. Shostack. *Threat Modeling: Designing for Security*. John Wiley and Sons, 2014.

[20] R. E. Smith. A contemporary look at Saltzer and Schroeder’s 1975 design principles. *IEEE Security & Privacy*, 10(6):20–25, 2012.

[21] W. Stallings and L. Brown. *Computer Security: Principles and Practice (3rd edition)*. Pearson, 2015.