Computer Security and Internet Security

Chapter 1: Basic Concepts and Principles

P.C. van Oorschot

June 10, 2018

Comments, corrections, and suggestions for improvements are welcome and appreciated.
Please send by email to: paulv@scs.carleton.ca

Chapter 1

Basic Concepts and Principles

Our subject area is computer and Internet security—the security of software, computers, 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 which host web sites, back-end servers which 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 which 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 successful 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, and which fall under the categories of *reliability* and *redundancy*, are complementary to the types of protection pursued by computer security.

The *fundamental goal* of computer security is to support computer-related services by means which provide or preserve security-related properties including the following:

1) *confidentiality*: the property of non-public information remaining accessible only to authorized parties, whether stored or in transmission (at rest or in motion). A common *technical* means is *data encryption*, involving 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. *Cryptographic checksums* are a data integrity mechanism for *detecting* integrity violations such as unauthorized data modification or deletion. *Prevention* is more difficult. Also important are hardware integrity and the integrity of people (consider bribery).

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

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

In discussing access control, the agents representing users, communicating entities, or system processes are called *principals*. A principal often has associated *privileges* specifying the resources it is authorized to access. The identity of a principal is thus important; asserted identities must be verified. This leads to the following two further properties, corresponding fundamental goals, and services which deliver or maintain them.

5) *authenticity*: the property of a principal, data, or software being genuine relative to expectations arising from appearances or context. *Entity authentication* provides assurances that the identity of a principal involved in a request is as asserted; this may support *authorization* (see above). *Data origin authentication* provides assurances that the source of data or software is as asserted; it is related to integrity (above), as modification by an entity other than the original source changes the source.

6) *accountability*: the state of principals being answerable for past actions, often achieved by recording which principals are responsible for actions and accesses. As the electronic world lacks conventional evidence like paper trails, visual records and human observation of events, evidence of transactions should be recorded by means which prevent principals from credibly denying (*repudiating*) commitments.

TRUSTED VS. TRUSTWORTHY. We carefully distinguish between the terms *trusted* and *trustworthy*. Something is trustworthy if it *deserves* our confidence, i.e., will reliably deliver against our security expectations. Something trusted *has* our confidence, whether deserved or not; so we rely on a trusted component, expecting security or other guarantees—but if it fails then all guarantees are lost.

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 is 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.

Computer security protects *assets*: information, software, hardware, and computing and communications services. Note that computer-based data manipulation allows control of many physical-world assets 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 the assets requiring protection, which specific users may access which assets and how, which security services are provided, and which system controls 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.

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 starts 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.

An *attack* is a deliberate action which, if successful, causes a *security violation*—such as control of a client device by an unauthorized party. Attacks typically exploit specific system characteristics or *vulnerabilities*, often software artifacts like misconfigurations,

unsafe default settings, and design or implementation flaws.[1] The source or *threat agent* behind an attack is called an *adversary* in theory, or an *attacker* in real-world systems.

A *threat* is any combination of circumstances and entities that may harm assets through a security violation. 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; 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 beg the questions: secure against whom, from what types of attacks?

A security policy allows a determination of 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, 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 may 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, is 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

[1] Ch.9 has Common Vulnerabilities and Exposures (CVE) system background and an example CVE entry.

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 particular threats are instantiated by attackers in a given period). V reflects the existence of vulnerabilities. C reflects cost or the impact of a successful attack. Equation (1.1) highlights the main elements in risk modelling 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.

Equation (1.1) is often 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 associated attack vectors exploit. 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 can be 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.[2] 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:

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

[2] To go into further detail: 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.

Here the sum is over all modeled security events of type 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 which are fundamental in not only 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 threat vectors?
4. What are the associated estimates of attack probabilities, or frequencies?

COST-BENEFIT ANALYSIS. The cost of deploying security mechanisms must always be taken into account. If the totals costs of a new security defence exceed the anticipated benefits (e.g., reductions in expected losses), then the defence is not justifiable 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.

RISK ASSESSMENT CHALLENGES. Quantitative risk assessment may be suited for incidents which 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 threat environments. Other barriers include:

- incomplete knowledge of vulnerabilities, worsened by rapid technology evolution;
- the difficulty of quantifying the value of assets such as strategic information, corporate reputation, and human time; and
- incomplete knowledge of threat agents and their *adversary classes* (see §1.5). 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.

***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.

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 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.

1.4 Design Principles for Computer Security

There is no checklist—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 1.6.3 discusses a type of checklist sometimes used in security analysis, but independently, security designers are encouraged to understand and follow a set of widely applicable design principles for security. We collect them all in one place here, and revisit them throughout the book with detailed examples to aid understanding.

P1 SIMPLICITY-AND-NECESSITY: Keep designs as simple and small as possible. Reduce the number of components used to those that are essential; minimize functionality, favour minimal installs, and disable unused functionality. Economy and frugality in design simplifies analysis and reduces errors and oversights. Config-

ure initial deployments to have non-essential services and applications disabled by default (related to P2).

P2 SAFE-DEFAULTS: Use safe default settings (since defaults often go unchanged). For access control, deny-by-default. Favour explicit permission (e.g., *white-lists* which list authorized parties, all others being denied) over explicit exclusion (e.g., *black-lists* which list parties to be denied access, all others allowed). Design services to be *fail-safe*, meaning that they fail "closed" rather than "open".

P3 OPEN-DESIGN: Do not rely on secret designs, attacker ignorance, or *security by obscurity*. Invite and encourage open review and analysis. Without contradicting this, leverage unpredictability where it brings no disadvantage; and note that arbitrarily publicizing tactical defence details is rarely beneficial (there is no gain in advertising to thieves that you are on vacation, or posting house blueprints). Example: undisclosed cryptographic algorithms are now widely discouraged; the *Advanced Encryption Standard* was selected from a set of public candidates by open review.

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 (has not been modified after being issued by the legitimate party).

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* (see later chapters); P6 and P7 have similar motivations. Restrict authorized cross-component communication to observable paths with defined interfaces to aid mediation, screening, and use as *choke-points*. Examples of containment means include: process and memory isolation, disk partitions, virtualization, software guards, zones, gateways and firewalls.

P6 LEAST-PRIVILEGE: Allocate the fewest privileges needed for a task, and for the shortest duration necessary. For example, retain superuser privileges only for actions requiring them; drop them until later needed again. This reduces exposure windows, and limits damage from the unexpected. It complements P5 and P7.

P7 MODULAR-DESIGN: Avoid designing monolithic modules that concentrate extensive privilege sets in single entities; favour object-oriented and finer-grained designs segregating privileges across smaller units, multiple processes or distinct principals. The complementary P6 guides where monolithic designs already exist, e.g., a `root` account should not be used for tasks when regular user accounts suffice.

P8 SMALL-TRUSTED-BASES: Strive for small code size for components that must be trusted, i.e., components on which the larger system strongly depends for security.

Example 1: high-assurance systems centralize critical security services in a minimal core operating system *security kernel*, whose smaller size allows more efficient concentration of security analysis. Example 2: cryptographic algorithms separate mechanism from secret, with trust collapsed down to a *secret key* changeable at far less cost than the more complex, non-secret algorithm.

P9 TIME-TESTED-TOOLS: Rely wherever possible on time-tested, existing security tools including protocols, cryptographic primitives, and toolkits, rather than designing and implementing your own. History shows that security design and implementation is a challenge even to experts; thus amateurs are heavily discouraged (*don't roll your own crypto*; *don't reinvent the wheel*). Confidence in methods and tools increases with the length of time they have survived (long-term *soak testing*).

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 for 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 which 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 which illuminate benefits, improve user experience, and minimize inconvenience. Mechanism viewed as time-consuming, inconvenient or without perceived benefit encourage bypassing and non-compliance. Example: a subset of Google gmail users voluntary use a two-step authentication scheme which augments basic passwords by one-time passcodes sent to a user's phone.

P12 SUFFICIENT-WORK-FACTOR: For security mechanisms susceptible to direct *work-factor* calculation, design so that the work cost to defeat the mechanism safely exceeds the resources of expected attackers. Use defences suitably strong to protect against anticipated classes of attackers (see *categorical schema*, discussed earlier).

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

P14 EVIDENCE-PRODUCTION: Record system activities through event logs and other means to promote accountability, help understand and recover from system failures, and support intrusion detection tools. Example: robust audit trails facilitate *forensic analysis*, to recover data and reconstruct events related to intrusions and

criminal activities. In many cases, means facilitating 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. Examples: *sanitizing input*, and *canonicalizing data* (such as fragmented packets, and encoded characters in URLs) address *code injection* and *command injection* attacks including *cross-site scripting* and *memory exploits*; important classes of attack can be mitigated by *type-safe* languages.[3]

P16 REMNANT-REMOVAL: On termination, remove all traces of critical information associated with a task, including remnants possibly recoverable from secondary storage, RAM and cache memory. Example: common file deletion removes directory entries, whereas *secure deletion* aims to make files unrecoverable even by forensic tools. Related to remnant removal, beware that while a process is active, traces 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 (*trust anchor* or *root of trust*). More generally, verify trust assumptions where possible, with extra diligence at registration, initialization, software installation, and other starting points in a lifecycle. Examples: *bootstrap code*, *trusted computing bases*, auto-updating software, *certificate chains*.[4]

P18 INDEPENDENT-CONFIRMATION: Use simple independent cross-checks to increase confidence in code or data, especially when potentially provided by outside domains or over untrusted channels. Example: the integrity of a downloaded software application or public key can be confirmed by comparing a locally-computed *cryptographic hash*[5] of that item to a "known-good" hash obtained from an independent channel such as a voice call, text message or widely trusted web site.

P19 REQUEST-RESPONSE-INTEGRITY: Verify that responses match requests in *name-resolution* protocols and other distributed protocols. Their design should verify consistency across steps, and detect message alteration or substitution, e.g., by cryptographic integrity checks designed to correlate messages in a given protocol run; beware protocols without 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 to confirm this.

P20 RELUCTANT-ALLOCATION: Be reluctant to allocate resources or expend effort, especially in interactions with unauthenticated, external agents that initiate an interaction. For processes or services with special privileges, be reluctant to act as a

[3] For details of examples, see later chapters.
[4] Again, for details of examples, see later chapters.
[5] See Chapter 2 for background on basic tools and support mechanisms from cryptography.

conduit extending such privileges to unauthenticated (untrusted) agents.[6] Place a higher burden of proof of identity or authority on agents that initiate a communication. (A party initiating a call should not be the one to demand: *Who are you?*)

P21 SECURITY-BY-DESIGN: Build security in, staring at the initial design stage of a development cycle—because secure design often requires core architectural support absent if security is an add-on or late-stage feature. Explicitly state the *design goals* of security mechanisms and what they are *not* designed to do—without knowing goals, it is impossible to evaluate effectiveness. Explicitly state all security-related *assumptions*, especially involving trust and trusted parties (supporting P17).[7]

P22 DESIGN-FOR-EVOLUTION: Have evolution in mind 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 modify designs as needed.

FURTHER NOTES ON DESIGN PRINCIPLES. Our principles overlap other ideas, which we relate here to the most relevant principles (rather than extend our formal list).

- SIMPLICITY-AND-NECESSITY (P1): simplicity and minimal deployments, and several other principles, support another broad goal: *minimizing attack surface*. Every interface that accepts external input or exposes programmatic functionality provides an entry point by which an attacker might change or acquire a program control path (e.g., install code or inject commands for execution), or alter data which 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 or restrict such access to authorized parties, and sanitize data input to them.

- SAFE-DEFAULTS (P2): a related recommendation for *session data* sent over real-time links is to *encrypt by default*[8] (which itself complements P4, as encryption mediates access to data). This motivates *opportunistic encryption*—encrypting session data whenever supported by the far 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.

- OPEN-DESIGN (P3): related is *Kerckhoffs' principle*—a system's security should not rely upon the secrecy of its design details.

[6] An example related to a *denial-of-service attack* is given in a later chapter.
[7] This differs from security policy, since policy need not necessarily specify assumptions.
[8] This is now understood to mean encryption with built-in data integrity, as discussed later.

- LEAST-PRIVILEGE (P6): related is the military *need-to-know* principle—access to sensitive information is granted only if essential to carrying out one's official tasks.
- MODULAR-DESIGN (P7): related is the financial accounting principle of *separation of duties*—overlapping 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.
- SMALL-TRUSTED-BASES (P8): related is the *minimize-secrets* principle—secrets should be few in number. One motivation is to reduce management complexity.
- TIME-TESTED-TOOLS (P9): underlying reasoning is that wide use of a heavily-scrutinized (thus less likely flawed) security mechanism is preferable to numerous independent novice implementations scantly reviewed; use of widely-used crypto libraries like OpenSSL is thus encouraged. Typically this overrides the older principle of LEAST-COMMON-MECHANISM—minimize the number of mechanisms (shared variables, files, system utilities) shared by two or more programs and depended on by all, motivated by *code diversity* potentially reducing negative impacts.
- The maxim *trust but verify* suggests that given any doubt, verify for yourself.[9] Design principles related to this line of defence include: COMPLETE-MEDIATION (P4), DATA-TYPE-VERIFICATION (P15), TRUST-ANCHOR-JUSTIFICATION (P17), and INDEPENDENT-CONFIRMATION (P18).

1.5 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 defences than a combined battalion of tanks and squadron of fighter planes.

ADVERSARY ATTRIBUTES. Important attributes of an adversary to consider include:

1. *objectives*—these often suggest target assets requiring special protection;
2. *methods*—e.g., attacker techniques, and anticipated types of attacks;
3. *capabilities*—computing resources (CPU, storage, bandwidth), skills, knowledge, personnel, opportunity (e.g., physical access to target machines); and
4. *funding level*—this often correlates with attacker determination and persistence.

Various schemas are used in modeling adversaries. A *categorical schema* classifies well-defined adversaries into *named groups* as given in Table 1.2.

[9] Given assertions by foreign diplomats whom you are expected to show trust in but don't really trust, the advised strategy is to feign trust while silently cross-checking for yourself.

	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 mass media uses the term *hackers*, which we prefer to reserve for computer system experts knowledgeable about low-level details.

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 hackers 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*.

We usually think of attacks launched from outside of an organization, by parties without any a priori special access to the target network; we call these *outsiders* and *outsider attacks*. In contrast, *insiders* and *insider attacks* originate from parties having some advantage over outsiders, e.g., internal employees who might thereby have physical access, or some network credentials as legitimate users (though not authorized to carry out fraud!). The line between outsiders and insiders may become fuzzy, for example when an outsider somehow gains access to an internal machine and uses it to attack further systems.

SECURITY EVALUATIONS AND PENETRATION TESTING. Some government departments and other organizations may require that prior to deployment in their organization, products be *certified* through a formal *security evaluation*. This process 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 government or international standards.

Another process, *penetration testing* (*pen testing*), involves customers or consultants they hire, using various customized tools to test software systems for vulnerability to specific exploits or classes thereof—typically related to known implementation errors in specific products, and known classes of misconfiguration errors. Product vendors themselves may carry out such testing at final development stages before product release, and as a complement to standard regression testing. Note that some design principles (above) hint at the types of errors to look for after-the-fact, should the principles be ignored.

SECURITY ANALYSIS. Our main interest is the type of *security analysis* commonly implied by the colloquial phrase "carrying out a security analysis". Its primary aim is to find vulnerabilities resulting from design flaws; a secondary aim is to suggest ways to improve defenses in case of (inevitable) failures. Security analysis is thus complementary to methods which find implementation and configuration flaws—it offers the potential to discover vulnerabilities not yet actively exploited, whereas penetration testing specifically exploits vulnerabilities in order to demonstrate them. Note that if we define the term *vulnerability analysis* to mean the process of identifying security weaknesses, then it includes both security analysis and penetration testing.

Security analysis involves considering a system's architectural features and components, identifying where protection is needed and the corresponding mechanisms intended to provide it, and walking through how existing designs meet security policy (system requirements). This must be done with specific thought to the target deployment environment, which implies also taking into account relevant assumptions and threats. The analysis should detail how existing or planned defence mechanisms address relevant threats and attacks identified by threat modeling, and which threats remain unmitigated. Thus the cornerstone of all variations of security analysis is *threat modeling* (Section 1.6).

RISK MANAGEMENT VS. MITIGATION. Not all threats can (nor necessarily should) be eliminated by technical means. *Risk management* combines the technical activity of estimating risk (Section 1.3), or often simply identifying threats of major concern (above), and the business activity of "managing" the risk, i.e., taking 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 de-commissioning the system.

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 which suggest things to look for, and to avoid, in the design of security-minded products and systems.

1.6 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, and anticipated. Those considered out of scope should be explicitly recorded as such. Threat modeling takes into account *adversary modeling* (Section 1.5), and should identify and consider all *assumptions* made about the target system, environment, and attackers. Threat modeling can be done by several different approaches, as discussed in the subsections below.

1.6.1 Diagram-driven Threat Modeling

A simple approach to threat modeling starts with a diagram for the target system to be built or analyzed. Draw an architectural representation showing system components and all communications links used for data flows between them. Identify and mark system gateways, guards, choke-points—wherever 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 the process by turning the architectural diagram into an informal *data-flow diagram*: trace the flow of data through the system for a normal task or transaction or service. Examining these flows, again ask: "What could break?" Consider other types of transactions or services, and eventually all tasks.

Next consider the *user work-flow*: trace through the actions of a user, from the time they begin a task to the time they end it. Begin with a common everyday task. Move on to less frequent tasks, such as installing, configuring and upgrading software (but also: abandoning or uninstalling software); account creation or registration (but also de-registering an account). Consider full *life-cycles* of data, software, users, accounts.

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 back-up 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 WiFi link in a café, hotel or airport—could that result in a man-in-the-middle scenario, 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 cleaning 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. Welcome to the fine art of threat modeling.

1.6.2 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 the overall attack goal (e.g., enter a house). Sub-nodes below it 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 sub-tree whose children specify ways of reaching it. Sub-trees end in *leaf nodes*. A path from a leaf node to the root lists the steps (the attack vector) composing one full attack; intervening nodes may detail pre-requisite steps.

The main output is an extensive (but usually incomplete) list of possible attacks. The attack paths can be examined to determine which 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 more important 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 or model, and in security analysis to cross-check that mechanisms are in place to counter all identified attack vectors, and to explain why particular vectors are infeasible for the relevant adversaries of the target system. Attack vectors identified may help determine which types of defensive measures are needed to protect specific assets from particular types of malicious actions. Attack trees can be used to prioritize vectors as high or low concerns, e.g., based on their ease, taking into account the relevant classes of adversary.

By default, multiple children of a given node are 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 (see above), 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 attack tree methodology encourages a form of directed brainstorming, adding structure to what may otherwise be an ad hoc task. The process benefits from a creative mind. It requires a skill which 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 approach is to draw a data-flow diagram showing a password's end-to-end paths, then identify points at which an attacker might try to extract information. An alternative is to build an attack tree, with goal to gain access to a user's account on a given system. A partial list of possible attacks follows.[10]

1. password guessing (both online and offline), including *dictionary attacks*

[10] Password-based authentication is discussed further in Chapter 3.

2. password capture, through:
   (a) visual observation of keyboard input or display output (e.g., *shoulder-surfing* by unaided eye, or recording devices via a smartphone or hidden camera)
   (b) non-visual *emanations* (electro-magnetic, acoustic) from displays, keyboards
   (c) intercepting wireless data (from a wireless keyboard or wireless data link)
   (d) intercepting wireline data, especially unencrypted transfer to remote sites
   (e) client-end malicious software (*keyloggers*, *screenloggers*, *Trojan horses*)
   (f) hardware keyloggers
   (g) server-side break-ins, both to web-facing servers and back-end databases
   (h) *man-in-the-middle* (MITM) or proxy-based attacks, including *pharming*
   (i) *social engineering* attacks, including *phishing*
   (j) client device theft, recovering stored passwords from memory or disk
3. reconstruction from partial information leaked on successive logins
4. software exploits that bypass the standard authentication mechanism
5. defeating password recovery mechanisms (e.g., guessing answers to weak *personal verification questions*; intercepting recovery passwords sent to email accounts).

Exercise (Free-lunch attack tree). Read the article by Mauw [8], for a fun example of an attack tree. As supplementary reading see *attack-defense trees* by Kordy [6].

Exercise (Recovering screen content). Build an attack tree with goal to extract data shown on a target device display (case 1—desktop; case 2—smartphone screen).

1.6.3 Other Threat Modeling Approaches: Check-Lists and STRIDE

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

Advantages of such check-lists are that extensive such check-lists exist; their thorough nature can help ensure that well-known threats are not over-looked by ad hoc processes; and compared to previously-mentioned approaches 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 check-lists risk becoming tedious, replacing a security analyst's creativity with boredom; and their length may dilute attention away from higher-priority threats. Check-lists 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 middleground is to use a small set of keywords to stimulate thought, without being overloaded by a longer list. A specific method following this direction is the *STRIDE* approach attributed to individuals at Microsoft. The term is a memory-aid for recalling 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 or past actions, often falsely;
- 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; and
- Escalation of privilege—obtaining privileges to access resources, typically referring to malware which 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 convenient in that most threats can be pigeon-holed into one of these boxes. The idea is to augment the diagram-driven approach by considering, at each point where the question is asked "Where can things break?", if any of these six categories of problems might occur. STRIDE thus offers another way to stimulate open-ended thought while staring at a diagram and trying to identify threats to its architectural components—in this case, guided by six keywords.

1.7 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.

1.7.1 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 adopt to changes in technology and attack capabilities. Model assumptions can also be wrong due to incomplete or incorrect information, naiveté, and failure to record assumptions explicitly—implicit assumptions are rarely scrutinized. Focusing attention on the wrong threats may mean threats of lower probability or impact than others. This can result from not only unrealistic assumptions but also: inexperience and lack of knowledge, failure to consider all possible threats (incompleteness), new vulnerabilities due to computer system and network changes, and truly novel attacks. It is easy to instruct someone to defend against all possible threats; anticipating the unanticipated is more difficult, as is predicting the future.

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. Assuming that an attacker aims to extract 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 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 the attacker has previously purchased at lower cost on his own account. The attacker 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 European online banks use *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. This prevents 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 website,

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.[11]

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 smart-phones 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 wrong. 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 deceptive factor is that the logic behind a security proof which requires some assumption A, may be valid even if A is false; the false premise invalidates the suitability of the logical proof to a particular system, but not the logic itself. A second is that a security model may truly provide a 100% guarantee that all attacks it considers are precluded by a particular security mechanism, while in practice the modeled system is vulnerable to attacks that the model failed to consider.

ITERATIVE PROCESS. At least as much art as science, threat modeling is an iterative process, requiring continual adaptation to more complete knowledge, new threats and changing conditions. As environmental conditions change, threat models that are not updated become obsolete, failing to accurately reflect reality.

Example *(Evolving Internet threat model).* Many Internet security protocols are based on the original *Internet threat model*. Its basic assumptions are: (1) end-points, 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 links in a hostile communications environment. However, assumption (1) is no longer generally valid in today's Internet where malicious software frequently compromises the integrity of end-point machines. For example, *keylogger* attacks remain viable.

Example *(Keyloggers).* Encrypting data between a client machine and server does not protect against common malicious software which intercepts keyboard input, to be relayed to remote 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.

1.7.2 Tying Security Policy back to Real Outcomes and Security Analysis

Returning to the big picture, we now pause to ask: How does "security" get tied back to "security policy", and how does this relate to threat models and security mechanisms? *Security mechanisms* are designed and used to support specific security policies. They

[11] Chapter 3 discusses one-time passwords and the related mechanism of *passcode generators*.

provide security services which can be grouped into the high-level categories discussed earlier: confidentiality, integrity, authorization, availability, authenticity, accountability.

OUTCOME SCENARIOS. Now consider outcomes related to security mechanisms:

1. The mechanisms fail to properly support the policy; the security goal is not met.
2. The mechanisms 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 mechanisms 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. The iterative process of security design and analysis 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 refine security requirements, shaping the security policy. The policy in turn shapes system design; security mechanisms which can support the policy in the target environment are selected. As always, key questions help:

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

Options available to mitigate past and future damage include not only attack *prevention* by counter-measures that preclude or reduce the likelihood that attacks successfully exploit vulnerabilities, but also *detection*, *real-time response*, and *recovery* after the fact (quick recovery can reduce impact). Consequences can also be reduced by insurance (cf. §1.3).

Once a system is designed and implemented, how do we post-evaluate it, or test that the protection measures work and that the system is "secure"? (At this point, you should be asking: What definition of "secure" are you using here?) How to test that security requirements have been met remains without a satisfactory answer. Section 1.5 mentioned security evaluation, penetration testing (often finding implementation and configuration flaws), and security analysis (often finding design flaws). Using check-list ideas from threat modeling, security testing can be attempted 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 if the system in place successfully withstands them. This of course leaves unaddressed unknown attacks, not yet foreseen or invented—by definition these are difficult to include in tests.

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 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 properties may change as even a small detail of one component changes, is updated, or reconfigured. A system's security properties are thus difficult to predict, measure, or see; we cannot observe security itself, but 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 occurring, or simply go unnoticed. The security of a computer system is not a testable feature, but rather is said (unhelpfully) to be *emergent*—resulting from complex interaction of elements that compose an entire system.

So then, what happens in practice? Evaluation criteria are typically determined from experience, and even the most thorough security testing is not believed to provide 100% guarantees. In the end, we seek to iteratively improve security policies, and likewise our confidence (*assurance*) that protections in place meet the requirements implied by policy. Assurance is provided by a combination of sound design practices, testing for common flaws and known attacks by standard techniques, formal modeling of components where suitable, ad hoc analysis, and relying heavily on experience. The best lessons come from attacks, and we learn from our mistakes—thus the *defender-attacker arms race*.

ASSURANCE IS DIFFICULT AND PARTIAL. In summary, for security assurance, we want not only to verify that expected functionality works as planned, but also that exploitable artifacts are absent. Security thus includes types of *non-functional goals* called *negative goals*. Overall security cannot be demonstrated, in part because mitigation of unknown attacks cannot be demonstrated; testing is largely possible only for known classes of attacks. Assurance is thus partial at best, and typically limited to well-defined scopes.

1.8 Why Computer Security is Hard

Many of the fundamental problems in computer security today strongly resemble those of many years ago, despite tremendous changes in computing hardware, software, environment and applications. For example, the first large-scale Internet security incident receiving widespread public attentions was the *Internet worm* of 1988. The technical details of the main attack vectors by which it spread were remarkably (or one may say, disappointingly) similar to those underlying the *Code Red* and related computer worms that emerged in 2001-2003.[12] The latter had larger overall impact due to the growth in population of Internet users. This points to the benefits of learning general principles of computer security—solid principles tend to remain true over time—but also highlights the difficulty that computer professionals have had in applying these principles to improve real-world security. The nature of computer security makes it challenging in practice, for numerous reasons. A number of these listed here now are explored in later chapters.

1. *intelligent, adaptive adversary*: while most science relies on nature not being capri-

[12] These and related malware incidents are the focus of Chapter 9.

cious, 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 helps geographically distant attackers (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 needs across diverse hardware-software and legacy systems delays and complicates deploying security upgrades, resulting in ongoing vulnerabilities even if updates are available.

10. *market economics and stakeholders*: market forces often hinder allocations that improve security, e.g., stakeholders in the position to improve security, or who would bear the cost of deploying improvements, may not be those who would benefit most.

11. *features trump 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 trumps 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 behaviour, 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 which 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 P21).

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! But rather than being depressed, as optimists we see a great opportunity in the many difficulties that complicate computer security, and in technology trends suggesting continued 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. We use computers, tablets and mobile phones every day to work, communicate, gather information, make online purchases, and plan travel. Our cars rely on software systems. (Does the idea of airplanes being controlled by software 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 software, computers and the Internet.

Perhaps the strongest motivation for individual students to learn computer security (and for parents and partners 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 for computer security problems. 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 ‡Endnotes and further reading

Additional barriers related to risk assessment and security metrics are noted by Jaquith [5, pp.31-36] and Parker [10], e.g., that 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 (see §1.8). For guidance on qualitative risk assessment, see risk assessment standards (e.g., NIST [9]) and textbook examples (e.g., Basin [3, §8.4], Stallings [17, §14.5]. Table 1.1 is based on a NIST guideline [9].

Core security design principles from 1975 by Saltzer and Schoeder [14] have been periodically revisited, for example, by Saltzer and Kaashoek [13, Ch.11] and Smith [16]; see also Basin et al. [3] for examples related to long-standing principles.

Lowry et al. [7] discuss adversary modeling. The research literature and older books discuss many *formal models* for security, including for specific policies related to confidentiality, integrity, and access control; for example, see Pfleeger [11, pp.245-263]. Gollmann [4, Ch.13] discusses formal security evaluation. Rescorla [12, p.1] discusses the Internet threat model. Attack trees are related to *threat trees* (see an early discussion by Amoroso [2, pp.15-29]), and predated by 1960s-era *fault tree analysis*. OWASP's "Top Ten" lists of security risks (vulnerabilities), specifically related to web application security, are supported by technical details. MITRE maintains the Common Attack Pattern Enumeration and Classification (CAPEC), an extensive list ("dictionary and classification") of security attacks intended to aid defenders; such extensive checklists, with supporting information possibly including attack instance prototype code, are sometimes called *attack libraries*. Shostack [15] gives an authoritative treatment of threat modeling, including STRIDE. Akerlof [1] explains the "market for lemons" and what happens when buyers cannot distinguish low-quality from (more costly) higher-quality products.

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

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

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

[6] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer. Foundations of attack-defense trees. In *Formal Aspects in Security and Trust (FAST 2010)*, volume 6561 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2011.

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

[8] S. Mauw and M. Oostdijk. Foundations of attack trees. In *Information Security and Cryptology (ICISC 2005)*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer, 2006.

[9] NIST. Special Pub 800-30 rev 1: Guide for Conducting Risk Assessments. National Inst. Standards and Tech., U.S. Dept. of Commerce, September 2012.

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

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

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

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

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

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

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

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