Computer Security and Internet Security

Chapter 1: Basic Security Concepts and Principles

P.C. van Oorschot

Feb. 18, 2017

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

***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 *encypt 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.