Chapter 3

Computer security

I have not written this book for the exclusive benefit of full time security researchers—my primary audience is rather one of technically-minded readers with a general computer science background who want to find out about ubicomp security. I shall therefore use this chapter to introduce some basic topics in computer security, in particular the three fundamental properties of confidentiality, integrity and availability, together with the cryptographic mechanisms that may help us protect them. (Despite the fact that this explanation dwells on those mechanisms so as to get the technical details out of the way, readers should heed the caveat of section 1.2 about security being more than just cryptography.)

For readers interested in a deeper treatment of such fundamentals, many excellent textbooks will provide further details. Gollmann [123] is one of the best all-round introductions to computer security; Amoroso [7] concentrates on the theoretical foundations of the discipline; Schneier [227] offers a thorough explanation of the cryptographic details, for which the monumental Kahn [149] provides the historical perspective; Anderson [11] beautifully illustrates how computer security and its numerous shortcomings affect the real world, while Rubin [223] gives you an extremely readable and up-to-date problem-oriented explanation of the security solutions you can adopt in practice. All of these texts are accessible to non-specialists, particularly the last two. More advanced textbooks include Koblitz [157], which explains the mathematical foundations of cryptography (particularly public key algorithms), and Menezes *et al.* [187], probably the best mathematical and algorithmic reference for anyone who intends to write crypto code.

3.1 Confidentiality

Confidentiality is the property that information holds when it remains unknown to unauthorized principals. The threat to confidentiality is known as **disclosure**.

The entity whose confidentiality needs protection typically assumes the form of a *message*, that is to say a sequence of bits *m*, that is transmitted from a sender *A* to a recipient[1] *B*. We wish the message *m* to be kept secret from any other entities that might (even legitimately) handle the communication between *A* and *B*.

3.1.1 Encryption and decryption

The principal mechanism to protect the confidentiality of *m* is *encryption*. A cipher is a pair of complementary algorithms, one transforming plaintext into ciphertext (*encryption*) and the other performing the reverse operation, called *decryption*. If we denote as *P* the set of all possible plaintexts and as *C* the set of all possible ciphertexts, then the encryption function is a bijection[2] from *P* to *C* and the decryption function is the inverse bijection from *C* to *P*.

Kahn [149, chapter 2, page 84], quoting Suetonius, reminds us that even Julius Cæsar used such a pair of algorithms: for him, the encryption function involved changing every letter of the message with the one three places down in the alphabet (so "a" would encrypt to "d"), and the decryption function was the obvious inverse. It is however apparent that, as soon as the enemy discovers the decryption function, the cipher becomes useless. One would then have to think of a new pair of algorithms (not that this would altogether be a bad idea if the original ones were as weak as Cæsar's).

A more flexible solution is to make the algorithms parametric: by introducing a key *K* as a parameter, each algorithm describes not one bijection but an entire family of them, and the bijections in a pair parameterized by the same key are of course still inverses of each other. Each bijection is a curried[3] version of the encryption or decryption algorithm with a specific key, and even if the enemy discovers the specific decryption bijection (i.e. key) used for a certain message, we can still safely use the same cipher as long as we choose another bijection pair by selecting a different key.

3.1.2 Security by obscurity (don't)

One of the cornerstones of modern cryptology is a set of principles established in 1883 by Kerckhoffs [155]. In essence, they explain that it is pointless to try to

[1] As Hamming [131] remarks, under many respects transmission in space (communication) and transmission in time (storage) are equivalent. The recipient *B* could just as well be *A* at a later time, so from this point of view even a file stored on *A*'s system may be viewed as a message.

[2] If you are not too sure what a bijection is (it's basically a one-to-one correspondence between two sets), you will find a review of this and related terms in appendix A. This also contains a brief cheat sheet (page 170) about letters such as $\forall, \exists$ and $\in$ that have been kicked out of shape by mathematicians.

[3] This term is explained in section A.4 on page 173.

hide the cipher algorithm and that all the security should reside in the choice of the key. The algorithm will become known eventually anyway, as it is impossible to guarantee the absence of leaks about the workings of the cipher when people other than the inventor have to use it. It is wiser to assume the algorithm to be public knowledge from the start, and at least reap the benefits of peer review by competent cryptologists.

Incidentally, this is the process that was followed by the American National Institute of Standards and Technology (NIST) to select the Advanced Encryption Standard (AES) cipher: 15 candidate algorithms were submitted by teams of cryptologists from around the world and were subsequently subjected to aggressive peer review from 1998 to 2000. In August 1999, a first round of selection chose 5 finalists [197], and in October 2000 the Belgian algorithm Rijndael created by Daemen and Rijmen [76] was announced as the winner [198]. This open and publicly reviewed selection process is probably the best possible guarantee of the quality of the chosen cipher and the absence of backdoors in it.

Contrast this with the deplorable practice of "security by obscurity", in which the algorithm is guarded as a secret (as was originally done for the infamous "Clipper chip"); this certainly makes the initial cryptanalysis harder for the adversary, but it may hide simple weaknesses that an open peer review process might have revealed—as the pay-TV industry discovered to its dismay after fielding tens of thousands of instances of a closed system that crackers soon learnt to bypass [213].

3.1.3 Brute force attacks

Once the cipher is public, the enemy who gets hold of an encrypted message can in theory decrypt it by applying all the decryption bijections in the family to it and selecting *a posteriori* the one which gives a meaningful result[4]. For this reason it is

[4]The alert reader will notice that this requires the nontrivial ability to recognize the meaningful result among all the meaningless ones. While a human reader will have no hesitation in flagging `AWEVN4@HBW-CUF` as a failed attempt and `MEET YOU AT 6` as a successful decryption, things become much harder when the detection has to be performed automatically (which is a necessity when exploring an entire key space) and particularly when the plaintext has no known structure. If the encrypted message contains some integrity-protecting redundancy, this can be checked automatically by the cryptanalyst. Otherwise, the cryptanalyst will have to look out for *cribs*, which are expected items of plaintext such as "Dear Sir", or for statistical properties of the plaintext. But the automatic application of such heuristics, as well as being time-consuming, can only give a probabilistic result. Besides, as the cardinality of the set of keys approaches that of the set of plaintexts (it actually *reaches* it in the one time pad, making the recognition theoretically impossible—see section 3.1.5), the solution space becomes dense, and heuristics that distinguish "plausible messages" from "garbage" stop working: `MEET YOU AT 6`, while in itself plausible, now competes with many more equally plausible plaintexts such as `MEET YOU AT 7`, `CAN'T SEE YOU TODAY` and even `EPIMENIDES SUGGESTS NOT TO BELIEVE THIS SENTENCE`. It must however be remarked that all these limitations apply only to the most difficult case for the cryptanalyst, the *ciphertext-only* attack. In many other situations it will

important to take into account the size of the family of bijections (i.e. the number of possible keys) when evaluating the safety of a cipher: if the key space is small enough that it can be exhaustively searched, the cipher will be weak regardless of any other consideration. (The reverse implication does not hold: a cipher may be weak due to design flaws, despite having a key space so large that a brute force search would take longer than the age of the universe.)

Under the strict export regulations of the United States known as ITAR[5] [253], for example, which were relaxed in January 2000 [252, 251], ciphers could be freely exported from the USA only as long as their key length did not exceed 40 bits. Even from the point of view of a not very resourceful attacker, any 40-bit cipher is weak, because a space of $2^{40} \approx 10^{12}$ keys can be searched by brute force in a reasonable time; if each decryption attempt takes 10 μs, the entire space can be searched in 10^7 seconds, which is about 4 months. Given reasonable assumptions, the ten-microsecond-per-attempt estimate is roughly in the correct ballpark for a modern personal computer implementing the decryption in software[6]; but if the attacker has more resources (such as dedicated hardware to perform the decryption) this estimate could go down by three or four orders of magnitude, bringing the time to try all keys to under an hour—perhaps just a few minutes.

One should also observe that brute force key search is ideally suited to parallel processing, since the problem space can trivially be divided into independent realms down to arbitrarily fine granularity. From March to June 1997, in an effort led by Verser, the idle time of about 70,000 machines across the Internet was harvested for key search; on the 96^{th} day this distributed system succeeded in cracking the first of the DES challenges sponsored by RSA Data Security [222].

DES, or the Data Encryption Standard, created by IBM in the early 1970s and adopted as a US FIPS standard in 1977 [107], is one of the most widely deployed ciphers in the world. No exploitable cryptographic weaknesses have been found in it (although its cryptanalysis prompted the discovery of new techniques such as

be possible to mount a *known-plaintext* attack (where the cryptanalyst knows for sure that a certain plaintext message p maps to a specific ciphertext c) or even a *chosen-plaintext* attack (where the cryptanalyst can feed a plaintext message to the encrypting bijection and observe the result—this might happen if the key and the encryption algorithm had been sealed in a tamper-proof enclosure that could thereafter be accessed freely). In these cases it is of course trivial to verify whether any decryption attempt yielded the correct result or not.

[5] International Traffic in Arms Regulations. ITAR mainly regulates war-grade weapons and parts for tanks, jets and submarines. But encryption software is ranked as "munitions" because of its military intelligence value.

[6] Gladman [116], who implemented all of the 15 AES candidates from scratch based on their published specifications [117], gives performance figures of 1389 cycles for key setup + 352 cycles for decryption of one block for the 128-bit key version of Rijndael (which was neither the fastest nor the slowest of the candidates, or of the finalists). On a 200 MHz Pentium Pro, which was the platform he used, this gives 8.7μs to decrypt one block with a new key. Of course this figure will vary with the computer used and the cipher to be cracked, but it gives us an order of magnitude estimate.

Biham and Shamir's *differential cryptanalysis* [34] and Matsui's *linear cryptanalysis* [185]), but it has a 56-bit key. At the time, this was asserted to be adequate for civilian purposes: indeed, in the US, DES used to be the only officially approved method for protecting unclassified data. Diffie and Hellman objected that this key was too short as early as 1977 [91], and in 1994 Wiener [262] presented a design sketch for a DES-specific hardware key search machine, which he estimated could be built for one million dollars and would search the whole space in 7 hours. Between 1997 and 1998 the Electronic Frontier Foundation [94] did actually build a highly parallel DES key search machine out of custom chips for 250 k$; in July 1998 this machine broke the DES II-2 challenge in 56 hours [95], finally settling the score on the limited security provided by DES even for civilian applications (any major company with valuable trade secrets to protect will expect its competitors to be able to afford a mere quarter of a million dollars for industrial espionage).

In November 2001, while this book was being copy-edited, Bond and Clayton [47, 72] announced their much cheaper (1 k$) FPGA-based DES-cracking machine that, in the context of a smarter attack, can recover all the DES and 3DES keys of an IBM 4758 running IBM's own CCA cash machine software. The 4758 is the "gold standard" tamper resistant cryptoprocessor (see section 6.2.2) and is widely used in the banking industry. This attack, which combines newly found weaknesses in the CCA software with a smart (i.e. non exhaustive) DES key search that can be run in just a couple of days, would allow a crooked bank manager to forge cash cards and PIN numbers at will and thereby raid the accounts of any of the bank's customers.

As part of the research that lead to this exploit, Clayton also compiled a valuable survey of brute force attacks [71].

3.1.4 The confidentiality amplifier

Having accepted that the security of the cryptosystem must reside in the secrecy of the key rather than in that of the algorithm, one is left with the problem of distributing the key itself. If *A* wishes to send a message *m* to *B*, but wants to encrypt it because she believes the communications channel between them to be insecure, how can she tell *B* which key to use for the decryption? Traditionally, the solution has been to transmit the key over another channel, deemed to be more secure; the diplomatic courier with an attaché case handcuffed to his wrist is a visual example of this.

In this scenario the cryptosystem acts as a "confidentiality amplifier": once bootstrapped with the help of the secure channel (which may be of insufficient bandwidth, of inadequate latency and of excessive cost to be used for the actual messages), the system ensures the confidentiality of a much larger and cheaper channel.

As noted by Diffie and Landau [92], another remarkable benefit of cryptography is that the robustness and cost of the confidentiality protection it extends over the communications channel do not in any way depend on the length or shape of the channel. Contrast this with other existing ways to secure a channel, such as embedding a data transmission cable inside an armoured and pressurized pipe.

3.1.5 Stream and block ciphers

A cipher of great theoretical importance is the **one time pad**, invented in 1917 by Vernam and Mauborgne [149], which works as follows. *A* wants to send message *m* (a string of bits) to *B*, so she generates a "pad" of completely random bits whose length is at least that of *m*. The ciphertext is obtained bit by bit, by XORing each bit of *m* with the corresponding bit of the pad, which acts as the key. It is trivial to verify that, given any bit p, XORing to it any other bit k twice returns the original bit p[7]. So, to decrypt, one simply XORs again each bit of the ciphertext $p \oplus k$ with the corresponding bit of the pad k; the two copies of the pad bit cancel each other out, yielding the plaintext p.

Since each bit of the pad is random and independent of the others, it contributes one full bit of entropy to the resulting ciphertext bit. (This is only true if the pad is never reused to encipher another bit—hence the "one time" in the name.) It is therefore theoretically impossible to recover the plaintext from the ciphertext without the pad, because any plaintext is equally likely. For every plaintext p and for every ciphertext c, there exists a pad k that will yield c if applied to p—namely the one obtained as $k = p \oplus c$.

The one time pad therefore offers perfect confidentiality with respect to a ciphertext-only attack. Its principal drawback is that the key needs to be as long as the message. If it were easy to send securely such a long pad to the recipient, one might prefer to send the message in the first place, instead of the pad. In other words, as a "confidentiality amplifier", the one time pad does not amplify very much[8].

There is a large family of ciphers that work like a one time pad but replace the pad with a pseudo-random number generator, whose sequence of pseudo-random bits k depends on an initial fixed-length key K. The theoretical unbreakability is lost, because the bits of the pad's replacement are no longer of maximal entropy. If the adversary can predict the pseudo-random sequence, the cipher can be broken. Ciphers like this, that work by XORing each bit of the plaintext p with the

[7] This is written $\forall p, k \in \{0,1\} : p \oplus k \oplus k = p$.

[8] This is not to say it is useless. It may still be useful to time-shift a secure transmission. You send the courier with the one-time tape now, which takes a day or two, and you can then phone the Kremlin later at no notice when something serious comes up.

corresponding bit of a "keystream" k, belong to the class of the **stream ciphers**[9]. A stream cipher $\mathrm{E}_K^{\mathrm{stream}}$ is a family of bijections from the infinite set S of all possible bit strings onto itself:

$$\mathrm{E}_K^{\mathrm{stream}} : S \to S.$$

It should be obvious that any stream cipher based on a keystream generator will fall to a *known plaintext* attack, i.e. one where the cryptanalyst knows that plaintext p encrypts to ciphertext c and wants to decrypt other ciphertexts encrypted under the same key. In fact, just as for the one time pad (which is itself a special case of stream cipher), the keystream[10] can be trivially recovered as $k = p \oplus c$.

Block ciphers were originally developed to counter this attack. A block cipher $\mathrm{E}_K^{\mathrm{block}}$ is a family of bijections from the finite set $S_{\mathrm{len}=b}$ of all bit strings of length b (the block size) onto itself:

$$\mathrm{E}_K^{\mathrm{block}} : S_{\mathrm{len}=b} \to S_{\mathrm{len}=b}.$$

Because it is a bijection of a finite set onto itself, a block cipher is a permutation of the set $S_{\mathrm{len}=b}$. In theory it could be implemented as a randomized lookup table, but in practice the table would be infeasibly large[11]. For this reason, the implementation is based on repeatedly stirring bits around in a somewhat more algorithmic fashion. But the point is that, because of their more general structure, block ciphers can be constructed so that even knowing plenty of (p,c) pairs for a given k is insufficient to discover the key itself.

If one needs to encrypt something longer than a block, there exist various *modes of operation* such as "cipher block chaining" to do so in a secure way with respect to the desired properties. Books such as Schneier [227] have all the details.

3.1.6 Public key cryptography

As we move towards a global communications infrastructure where correspondents may be separated by thousands of kilometres, it becomes harder and harder to secure the lower bandwidth channel required for key distribution using physical means. The revolutionary invention that allows keys to be distributed securely over an insecure channel is *public key cryptography*, introduced by Diffie and Hellman in 1976 [90] (the British spy agency CESG claims to have discovered the principle in 1970, but didn't tell anyone until much later [98, 268]). Like many of the greatest ideas, its elegant simplicity allows it to be explained in only a few sentences.

[9] There also exist stream ciphers not based on pseudo-random keystream generators—the rotor machines used in World War II [149] being a notable example.

[10] Though not the original key K.

[11] Even for DES, with a block size of 64 bits, the table would have to hold 2^{64} words of 64 bits each, i.e. 2^{72} bytes, i.e. 4 billion terabytes. And this is still peanuts compared to the equivalent figure for AES, whose block size is at least 128 bits.

As before, the cipher is a family of pairs of bijections, with the elements of each pair being the inverse of each other. Now, however, the bijections in a given pair are not indexed by the same key: there is one key to select the encryption bijection and another one to select the decryption bijection, and it is not feasible to derive the decryption key from the encryption key[12]. Each prospective recipient of secret messages chooses her own pair of bijections and makes her encryption key public, while keeping the decryption one secret. Suppose *B* wants to send a confidential message to *A*: he can encrypt the plaintext using the encryption key that *A* made public. This will yield a ciphertext that can only be decrypted by *A*'s secret key, that she never revealed to anyone else. Nobody other than *A* will be able to read the encrypted reply—not even *B* who created it. If *A* wishes to reply, she needs to encrypt her answer under a different key—*B*'s public key in the above case. This construction removes the necessity for a confidentiality-protected channel to transmit the keys: the secret keys never leave their owners, while the public keys, as their name implies, can be distributed to anyone without ill effects. If an eavesdropper acquires *A*'s key, all that he will be able to do with it is to send encrypted messages to her.

There is one remaining vulnerability of this scheme, linked to the fact that *B* could be tricked into accepting a key as belonging to *A* when in fact it was manufactured by an active eavesdropper. We shall examine this important point in detail when we discuss man-in-the-middle attacks in section 3.4.3. For the moment, let us simply remark that this same **key distribution problem** exists to an even greater degree in the context of conventional, symmetric-key cryptography, and that it is not a vulnerability *introduced* by the public key construction.

Diffie and Hellman thought of the principle of public key cryptography, but could not at the time suggest an implementation with all the properties they described—this came a couple of years later with the cipher proposed by Rivest, Shamir and Adleman [219], later to be known as RSA. What the two original authors *did* suggest was a "key agreement" scheme, now appropriately known as Diffie-Hellman, allowing two parties to establish a shared secret over a channel subject to eavesdropping. This secret could then be used as the key for a symmetric cipher.

3.1.7 Hybrid systems

Under the hood, public key ciphers are built in a completely different way from their symmetric key counterparts. They require mathematical structure in order to provide the properties of a public key system: the public and the private keys must be mutual inverses, but it must be impossible to derive the latter from the former. To

[12] It is however feasible to generate a pair of keys that correspond to mutually inverse bijections.

this end, problems from number theory, based on modular arithmetic over numbers that are several hundred digits long, are used as the core of the cipher. To go into details would lead us well outside the scope of this book; what we want to remark here is that to encrypt or decrypt a given message with a public key cipher is about 1000 times more computationally expensive than with a symmetric key cipher of comparable strength[13]. What is then used in practice is a *hybrid cipher*: the sender generates a random session key, encrypts it under the recipient's public key and transmits it; the rest of the traffic is then encrypted with a symmetric cipher under the session key that the two parties now share. This combines the key management convenience of the public key cipher with the efficiency of the symmetric cipher (at least asymptotically).

It is interesting to observe that this sort of hybrid cipher is no longer a bijection—in fact it's not even a function any more. Since the session key is randomly chosen at each encryption, encrypting the plaintext P under the public key K will yield a different ciphertext every time. (Fortunately, of course, all these ciphertexts will still decrypt to the same plaintext P.) An advantage of this arrangement is that known ciphertext attacks become harder. It is no longer possible for the cryptanalyst who guesses the plaintext to verify the validity of this guess by encrypting it under the target public key, because even the correct guess would encrypt to a different ciphertext than the one observed, unless the random session key were the same. Note the conceptual similarity between this situation and the technique of "salting" described in section 3.4.1.

3.1.8 Other vulnerabilities

Before closing this introductory section on confidentiality-protecting mechanisms we should emphasize that the cost of the brute force attack on a cipher is only an *upper* bound on the cost of breaking the code, sufficient to dismiss a cipher as insecure but insufficient to promote it as safe. Especially in the case of a home-grown cipher, cryptanalysis is likely to find other weaknesses. More importantly, though, from a systems point of view the cipher is rarely the weakest point, and most breaches of confidentiality exploit other weaknesses such as protocol failure,

[13]To compare the strengths of two ciphers, one must in fact compare the effort it takes to break them. For a properly designed symmetric cipher with no known shortcut attacks, the effort is that of a brute force search, which is proportional to the cardinality of the key space. With a public key cipher, instead, one does not try all possible private keys but rather tries to derive the private key from the public one. This is of course laborious, by design, and typically proved to be as hard as some well-known difficult mathematical problem, but still not as bad as exhaustive search. For this reason, the key lengths for a public key cipher will be much longer than those for a symmetric cipher of comparable strength. Schneier [227, section 7.3, table 7.9] lists pairs of key lengths of similar resistance for symmetric and public key ciphers.

inappropriate key management, implementation defects (e.g. poor random number generation), physical vulnerabilities and so on.

3.2 Integrity

Integrity is the property that data holds when it has not been modified in unauthorized ways. There does not seem to be a generally agreed upon term in the literature to indicate the threat to integrity; it feels natural to me to suggest **corruption**.

As we saw, protecting the *confidentiality* of a message *m* in transit on a communication channel between *A* and *B* means ensuring that nobody other than *A* and *B* can discover the contents of *m*. Protecting the *integrity* of *m* under the same circumstances means ensuring that, once the message leaves *A*, nobody can alter it until it reaches *B*. In practice it is impossible to prevent an attacker who has control of the channel from altering the message, so what we actually mean is "ensuring that nobody can alter *m* without *B* noticing".

3.2.1 Independence from confidentiality

One might superficially think that confidentiality implies integrity: if you change one bit of a ciphertext *C*, the modified ciphertext will certainly decrypt to garbage and the recipient will notice. Actually this is not always so, and this is particularly evident for additive stream ciphers, where the ciphertext is obtained by the bit-by-bit exclusive-or of the plaintext and a pseudo-random key stream. If the attacker knows the exact format of the message, he will know which bit positions correspond to a specific field. If he also knows or guesses the plaintext, or at least the relevant part of it, for example the second field in "I hereby transfer <*unknown amount of money*> to <*John Smith's bank account number*>", he may manipulate the ciphertext to yield the desired plaintext, such as "I hereby transfer <*amount of money still unknown, but who cares*> to <*the crook's own bank account*>" . The simple algebra behind this is

$$guessedPlaintext \oplus unknownKeystream = knownCiphertext$$

from which the attacker extracts *unknownKeystream*, and

$$alteredCiphertext \oplus nowKnownKeystream = desiredPlaintext$$

from which the attacker works out which *alteredCiphertext* to substitute in the message in transit on the channel. This is called an *attack in depth*. Let us therefore remember that, in general, neither integrity nor confidentiality implies the other.

Do not take this just as an academic warning. The mistaken belief that confidentiality implies integrity is a genuine problem, and people get this wrong all the

time, silly as this may seem to you now after having read the trivial boolean algebra in this section. A high-profile culprit, which misused a stream cipher for integrity purposes, is the 802.11 wireless LAN system popularly known as Wi-Fi. This was first pointed out by Borisov, Goldberg and Wagner [48] (see section B.8).

3.2.2 Error-detecting codes

The core idea of integrity protection is to transmit a more robust message by augmenting the payload with some appropriate redundancy which the recipient can check in order to detect modifications. This framework aptly describes the error-detecting codes such as CRCs (cyclical redundancy checks) that should be familiar from information theory [131], but here we have an extra twist: the errors we are trying to detect may be caused not only by random noise but also by malicious forgery. A code that will detect bit errors in 99.9999% of the cases is probably good enough for many communications applications, assuming that bit errors happen randomly; from the security point of view, however, we are in a different scenario: the attacker will *actively search* for that specific 0.0001% of bit errors that the code cannot detect, looking for any that he might turn to his advantage. Suddenly, the probability of failure becomes much higher than 0.0001%.

3.2.3 Hash

An error-detecting code with an external interface similar to that of a CRC, but suitable for integrity protection, is the *cryptographic hash function*, sometimes indicated as *one-way hash* to convey the intuitive meaning that it is easy to compute the function in the forward direction, but practically impossible to compute it in the reverse direction. It is a surjection[14] from the infinite input set S of all possible bit strings to the finite set $S_{\text{len}=n}$ of all possible bit strings of length n, where n is the output size of the hash function (e.g. 160 bits for the Secure Hash Algorithm SHA-1).

$$\mathrm{h} : S \to S_{\text{len}=n}$$

Its fundamental property is *non-invertibility*: given any hash output $y \in S_{\text{len}=n}$, it is computationally infeasible to find an input $x \in S$ such that $\mathrm{h}(x) = y$ (despite the fact that there will usually be an infinite number of items x with this property). The idea here is to produce a representative "fingerprint" of any input message; this way, if the hash output is secure from modifications, it will not be possible for the attacker to modify the message in a way that still matches the hash.

[14] Well, at least conceptually it is. Theoretically, though, there is no guarantee that *all* elements of the domain will have a preimage. Most elements of $S_{\text{len}=n}$ will have infinitely many preimages, but there could be lonely elements of $S_{\text{len}=n}$ that no input string generates.

An even stronger requirement for a hash function is *collision resistance*[15]: it is infeasible to find two inputs with the same image (no matter what it is). If one could do that, then it would no longer be possible to consider hashes as representative fingerprints of longer strings, and "birthday attacks" such as the one famously described by Yuval [269] would become possible. Without going into details, that's where someone makes you sign an unfavourable document while making you believe that you are signing a favourable one, thanks to the fact that they both have the same hash.

We recognize the hash as an "integrity amplifier" in the same sense as we saw the cipher as a "confidentiality amplifier" in section 3.1.4: securing the integrity of a small bit string, for example by publishing it as a line ad in a newspaper, has the effect of similarly protecting an arbitrarily long message, like a 10 MB transaction log that you published on your web page.

One of the canonical warnings that accompany the introduction of error-detecting codes is that we cannot expect errors to concentrate only on the payload, ignoring the added redundancy. The equivalent observation in the security domain highlights that the hash works as an integrity amplifier only given a high integrity channel for bootstrapping purposes; if the attacker is able to modify the hash as well as the payload, it will be trivial for him to calculate a new hash that matches the modified message. There exist however two other cryptographic primitives, the MAC and the digital signature, capable of withstanding that type of attack.

3.2.4 MAC

The MAC, or *message authentication code*, is a bit like a hash parameterized with a secret key—each key you apply to the MAC gives you a different hash function. We saw a similar situation with ciphers in section 3.1.1: the hash is a surjection, while the MAC is a family of surjections. You might conceptually view the hash as a MAC whose key has been curried away with a well-known constant[16].

Both sender and recipient need that key: the former to calculate the MAC from the payload in the first place, the latter to recompute it for verification. Now the attacker who wants to modify the message can no longer generate a new MAC for the forgery, because he does not know the key. For the same reason, he cannot even *check* whether his forgery matches the old MAC. So this solution does away

[15] Commonly, but less accurately, also referred to as *collision-freedom*. The literal interpretation of this locution is of course an impossibility given the cardinalities of the input and output sets.

[16] What happens in practice is usually the reverse: the MAC is implemented from the hash by combining the key and the payload in a complicated way, as in the HMAC construction [162]. Another way to obtain a MAC from a hash is to encrypt the hash value using a symmetric cipher—this is simpler to understand but may be more costly to implement in terms of code size and running time, because one uses both a hash and a cipher.

with the need for the low bandwidth, high integrity channel—but note that it now requires a mechanism to allow sender and recipient to establish a shared secret key[17].

3.2.5 Digital signature

The digital signature is a development of public key cryptography. It attempts to recapture some of the defining properties of the written signature, namely that (at least in theory) nobody other than the signer may generate it, and anybody can verify its validity. Additionally, to avoid cut and paste attacks that are trivial on bit sequences even if they take some effort on paper artefacts, the digital signature must depend on the document being signed.

The construction originally proposed by Diffie and Hellman [90] to achieve this is simple and elegant, and was invented before a mathematical primitive to implement the underlying cipher could be found (the solution to that came two years later thanks to Rivest, Shamir and Adleman [219]).

Assume the availability of a public key cipher, as described in section 3.1.6 above: user A publishes a public key K_A and holds a private key K_A^{-1} known only to herself. Assume furthermore that $P \equiv C$, that is to say that the set P of all plaintexts coincides with the set C of all ciphertexts, so that it is possible to apply the decryption bijection to elements of P. It is then possible for A to produce a "signed" version of message $M \in P$ by exhibiting the following pair:

$$M, \mathrm{D}_{K_A^{-1}}(M)$$

which consists of the message accompanied by its image through the decryption bijection[18]. Nobody other than A knows the decryption bijection, so nobody else could have generated the signature; at the same time, though, anyone can check its validity by applying the publicly known encryption bijection E_{K_A} to $\mathrm{D}_{K_A^{-1}}(M)$, and verifying that it yields M.

This core idea is susceptible of refinements. For a start, there are clear performance advantages[19] in signing $\mathrm{h}(M)$ rather than M. It is also beneficial to keep a

[17] Not as bad as it sounds, because the key can be reused many times to protect the integrity of different messages. So we are not simply trading the requirement for a low bandwidth, high integrity channel with that for a low bandwidth, high confidentiality channel, but rather with the requirement for a *once-only* high confidentiality channel. Maybe not even that, if the shared secret were established through a key agreement scheme like Diffie-Hellman. However, the integrity properties of the mechanism used to establish the shared secret, whether it is the confidential channel or the DH exchange, must (almost recursively) be taken into account.

[18] Recall that, as explained in section 1.4, my notation for decryption is $\mathrm{D}_{KEY}^{\mathrm{CIPHER}}(ciphertext)$.

[19] Without loss of security as long as it is infeasible to find collisions for the hash function.

clear separation between encryption and decryption on the one hand, and verification and signature on the other, despite the fact that the above construction shows that the second pair of functions can be implemented with the first. Interesting threats come about when a user makes no distinction between signature and decryption, from the "chosen protocol attack" (someone runs a challenge-response protocol with you, as in section 3.4.3, asking you to decrypt something that they claim is a random number encrypted under your public key, but in fact they are making you sign a message saying that you'll pay them lots of money) to the risks of legal seizure (the police force you to reveal your secret key under court order to decrypt messages you received, and can now forge your signature on fake incriminating documents[20]). A convenient way to enforce the separation is to have different key pairs for encryption/decryption and for signature/verification. There are also algorithms of a different design where signature is implemented without resorting to decryption.

3.2.6 Integrity primitives compared

Whenever I explain all this, I find it useful to summarize the differences between these three integrity-protecting constructions by lining them up in a matrix[21] like table 3.1. The main distinction between the various primitives stems from identifying who can generate the code and who can verify it. This in turn determines the suitability of the construction for a given purpose, and one could add further columns to list what the extra requirements would be (e.g. a shared key in place) to achieve given goals.

	Who can generate it	**Who can verify it**
Hash	Everyone	Everyone
MAC	Holders of secret	Holders of secret
Signature	Holder of secret	Everyone

Table 3.1. A comparison of integrity-protecting primitives.

A subtle point highlighted by this comparison regards the transferability of the beliefs held by the verifier of the code. Assume that B receives a pair (m, c) over

[20] This cuts both ways, and to some principals this will be a feature, not a bug, in so far as it provides them with *plausible deniability* [221].

[21] The æsthetically-minded reader, seeking an elusive symmetry, may now be wondering about the missing fourth combination in the table—the one describing an item that everyone can generate and that only the holder of the secret can verify. A moment's thought shows that, with some flexibility in the interpretation of "verify", this primitive is simply public key encryption. It does not appear in the table because it is not an integrity primitive.

an insecure channel; *m* is a message purporting to be from *A*, while *c* is an integrity protecting code; assume furthermore that, upon verification, *B* finds that *c* matches *m*. What can *B* deduce about the genuineness of *m*, and can he transfer this belief to a third party *C*?

In the case of the hash, *B* has no guarantees at all: the message could be a fake and the hash might have been recalculated by the forger. (The hash is only useful when its own integrity is guaranteed.)

In the case of the MAC, if *B* is correct in assuming that only he and *A* know the secret key *K*, then he can deduce that the message is genuine. However—and this is the interesting bit—he cannot transfer this belief to anyone else. He certainly cannot convince an outsider *C* that the message is genuine: *B* would have to divulge the key to allow *C* to check the MAC, but since *C* does not know the real *K*, as far as he is concerned *B* could have easily made up both message and key, and calculated a new MAC before showing the pair to *C*. So the fact that the MAC checks proves nothing to *C*.

But there's more: even if we assume that *A*, *B* and *C* all shared the secret key *K* from the start, it is still impossible for *B* to prove to *C* that the message he is showing has not been changed from what *A* sent. Since *C* gets both message and MAC from *B*, *B* could still have made up the message and recomputed the MAC with the common key. The same holds in reverse: in fact, as soon as the key is shared by three parties instead of two, even *B* himself can no longer be sure that the message he received from *A* was not modified in transit by *C*.

The situation changes with the digital signature: the key that *A* uses to sign messages is never shared with anyone, since all that is needed for verification is the public key. Therefore, if the signature matches the message, it must perforce have been produced by *A*[22], and this argument is just as convincing to the first recipient *B* as to any subsequent principal who obtains the signed message.

In other words, unlike what happened with the MAC, *B* can show message and signature to a third party *C* and convince him that the message is the one originally generated by *A*. This property of digital signatures is often indicated as **non repudiation** to indicate that, once *A* signs a message, she cannot later deny to have done so, because nobody else could have generated that signature.

[22] We are sweeping under the carpet a number of problems that may occur in the real world, such as *A* temporarily "losing control" of her purse with the signing smartcard; but for this first order description let's stick to the idealized behaviour where principals are assumed to be able to keep keys secret.

3.3 Availability

Any system that performs its advertised service in a timely fashion when requested to do so by an authorized user enjoys the property of **availability**. The threat to availability is called **denial of service**.

An influential analysis of the problem of denial of service is due to Gligor [118, 119], who introduced the concept of **maximum waiting time** (MWT). The system should advertise the intended MWT for each of the services it offers. Whenever a user who issued a legitimate request has to wait for longer than the corresponding MWT, that user is being denied service.

Gligor observes that, while confidentiality and integrity are essentially concerned with what a user is allowed to do ("May she read? May she write?"), availability is concerned with what the authorized user is actually *able* to do ("Okay, she is allowed to write, but will it work if she tries?"[23]).

3.4 Authentication

Authentication is the process of verifying a principal's claimed identity. It is the logical step that follows **identification**, i.e. establishing who that principal claims to be. Both steps are necessary to convince the verifier of the identity of her interlocutor. (The familiar two-step interrogation of "userid? password?" is the perfect illustration.) It should be apparent that identification is the easier activity of the two (one just listens), while authentication is the one where some detective work is required of the verifier.

Authentication is by necessity a frequent event in distributed systems. Every day, many times a day, a computer M in your organization will receive an access request from an entity claiming to be legitimate user A, and M will have to decide whether to accept or reject the request based on its assessment of whether the bits that come out of the channel are sufficient proof that the principal at the other end really is A.

A basic taxonomy of authentication methods is neatly summarized in the traditional suggestion to check "something you know, something you have or something you are", examples of which might respectively be a password, a passport and a fingerprint. This classification is not exhaustive: Gollmann [123], for example, also lists "what you do" (e.g. keystroke patterns) and "where you are" (potentially of interest for ubiquitous systems). Imaginative readers may come up with further suggestions.

[23] Language purists will delight at the chance of being able to express this as: "She may write, but can she?".

As far as this introductory survey is concerned, though, we shall concentrate on the "something you know" approach: verifying knowledge of a shared secret is by far the most widely used foundation for authentication in computer networks today.

3.4.1 Passwords

Let us therefore assume that a principal A wishes to authenticate herself to a machine M; A is known to M by her name "A" (this could be the login name) and can prove her identity by demonstrating knowledge of a secret password p that she previously agreed with M.

The simplest way to demonstrate knowledge of the password p is simply to say it. This is what users normally do to log into a local computer.

The problem here is that, if M keeps a list all the passwords of its users, this password file becomes a valuable target for an attacker. Encrypting the file would bring little benefit, since M itself would need the key to decrypt it in order to be able to check any supplied password; so the attacker who managed to break into M could simply steal both the file and the key.

To address this, M may record a hash of A's password instead of the password itself—a technique pioneered by Needham at Cambridge [263] in the 1960s, later adopted by UNIX and now standard on most modern systems. This means that an attacker grabbing the list stored on M does not have access to any passwords.

But, despite this improvement, the system still isn't invulnerable: a more sophisticated adversary will mount a **dictionary attack**. Banking on the fact that many users will choose passwords that are easy to guess (e.g. names of loved ones, swear words, lines from favourite songs or movies ...), the attacker generates hashes of such candidate guesses and checks whether the result matches any entry in the list stolen from M. If it does, a password has been found. The guesses may even be precomputed once and for all and stored in a lookup table indexed by the hash value; then the attacker can crack on sight any account that used one of the guesses as its password. Safeguards against this kind of attack, other than educating users about the choice of proper passwords, include salting (see below) and artificially slowing down the hash function in order to increase the attacker's workload. Morris and Thompson [195] discuss those issues in detail, while a recent study by Yan *et al.* [267] provides experimental results on the (poor) choice of passwords even by users who have received specific training.

Salting is the practice of adding random bits to the user-supplied password before hashing it. These random bits have to be stored on M in the password list next to the hash value, otherwise the system itself would not be able to regenerate the hash for comparison even when supplied with the correct password. So this technique does not prevent the attacker from verifying the validity of a guess, but it does frustrate precomputation and parallel search. Furthermore, two users might

choose the same (poor) password, e.g. “samantha”; without salting, their hash values would be identical, telling an attacker that the penetration of one account also opens the door to the other, and also highlighting that the password is so easy to guess that those two users thought of it independently. Even more interestingly[24], if one of the users with the poor password noticed this circumstance, it would be trivial for him to abuse the other user’s account.

Whether A exhibits p or $\mathrm{h}(p)$, if this happens in cleartext across the network (as in Telnet, Rlogin, FTP and in the HTTP Basic Authentication that most web sites use to password-protect a page) then an eavesdropper can record what A says and replay it later to impersonate her. Note that the hash offers no protection here: the attacker won’t know the actual password, but won’t need it to impersonate A. To defeat this **replay attack** it is necessary to ensure that A’s authenticators can not be predicted from previous ones. This implies, among other things, that they will have to be always different.

3.4.2 One time passwords

An elegant way to do this using a chain of hashes was originally devised by Lamport [167], then implemented as a product (S/KEY) by Bellcore [130] and finally released as an open specification via a series of Internet RFCs [127, 128, 129]. The essence of the idea is for A to generate a series of passwords $p_0, p_1, \ldots, p_n$ linked by the recurrence relation

$$p_i = \mathrm{h}(p_{i+1}).$$

These are computed by A in reverse order, starting from a randomly chosen value for p_n. A keeps the entire list to herself and bootstraps by securely sharing p_0 with M. At the first authentication, A presents p_1 to M. M doesn’t know p_1, but he can verify that it is genuine by checking whether $\mathrm{h}(p_1) = p_0$. This procedure extends by induction to all other passwords in the chain: A just shows p_i at the i^{th} authentication, and M will know its hash p_{i-1} from the previous round. Eavesdropping attacks are defeated and so are attempts at stealing the “password file” from M. The information held by the server refers only to the past: it cannot be used to reconstruct the portion of the chain that has not yet been used, and therefore it can be used for verification but not for impersonation.

Among the disadvantages of the scheme are the need to “reload” after n authentications, the need for A and M to maintain synchronization (the protocol is not stateless) and the need for A to store the whole chain of passwords (though, of course, used passwords can be thrown away). This last requirement might be

[24] Taking insider fraud into account is still, to some extent, a paradigm shift. Needless to say, this is a problem. Major security holes go undetected for a long time because of this mindset.

traded off against extra computation in the unlikely case that A prefer to recalculate the chain from p_n at each authentication.

3.4.3 Challenge-response and man-in-the-middle attacks

Authentication can be performed in a stateless manner by using a **challenge-response** strategy. M sends A a random number n which A must return encrypted under the shared secret key K_{AM}.

$$\begin{aligned} M \to A: &\quad n \\ A \to M: &\quad \mathrm{E}_{K_{AM}}(n) \end{aligned}$$

Note that in this case a MAC, which is not invertible, would serve the same purpose as the encryption, since there is no need ever to decrypt the result; to verify, M simply performs the same encryption and compares the result, as per Needham's technique described in section 3.4.1.

This is the model originally used for air force IFF (identify friend or foe) systems, which led to the development of block ciphers. Jet fighters move at such speed that it is not possible to rely on visual identification to decide whether the dot on the radar is one of ours that should be protected, or one of theirs that should be shot down. If the fighters from the same side share a secret key, they can challenge each other by radio using the above protocol, and fire at those who can't prove ownership of the key.

The method is safe from passive replay attacks, but it remains vulnerable to more sophisticated *active* attacks, such as "man-in-the-middle". Anderson [11, section 2.2.2] relates of such an attack in the course of a battle between the South African and Cuban air forces at the border between Namibia and Angola and, with characteristic wit, labels it "MIG-in-the-middle". (The story later turned out to be apocryphal, but it still makes the point nicely.)

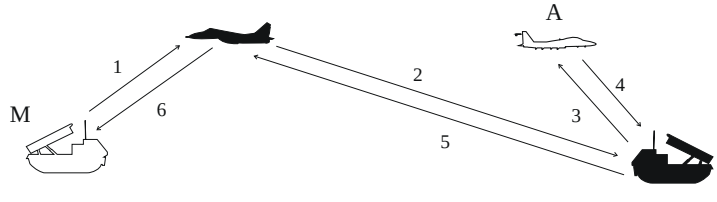
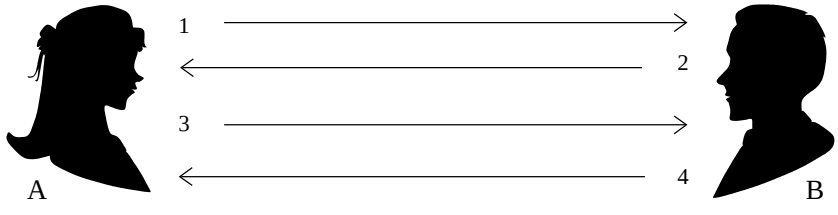


Figure 3.1. Anderson's "Mig-in-the-middle" attack.

Assume that the white side has an aircraft A and an anti-aircraft gun M that share a key. When M sees any aircraft it challenges it, and fires at it unless it responds correctly. The black side waits until the white plane attacks black territory

and at that point it sends a black plane to raid the white headquarters (figure 3.1). The black plane gets challenged (1) by the white gun *M*, so it relays (2) the challenge to the black anti-aircraft gun that spotted *A*; the black gun, in turn, challenges (3) *A*, who provides (4) the correct response. This is relayed (5) back to the black plane, who answers (6) *M*'s challenge correctly, is let through unharmed and proceeds to carpet-bomb the white headquarters.

Similar attacks apply to many other situations, such as the one described in section 3.1.6 where two users who do not share a secret establish a confidential communications channel using public key cryptography. Here *A* and *B* are the honest players, who believe they are having the exchange pictured in figure 3.2, which is safe from passive eavesdropping attacks.

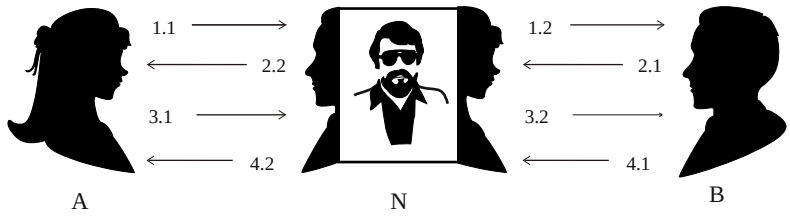


(1)	$A \rightarrow B:$	$K_A;$	*A* enables *B* to send secret messages to her.
(2)	$B \rightarrow A:$	$K_B;$	*B* returns the favour.
(3)	$A \rightarrow B:$	$\mathrm{E}_{K_B}(m_A);$	*A* sends secret message m_A to *B*.
(4)	$B \rightarrow A:$	$\mathrm{E}_{K_A}(m_B)$	*B* replies with secret message m_B.

Figure 3.2. What *A* and *B* think is happening.

Unfortunately for them, the malicious network operator *N* intercepts their messages and substitutes them with forgeries. He manufactures two fake key pairs, $(K_\alpha, K_\alpha^{-1})$ and (K_β, K_β^{-1}), the first to pretend to *B* that he is *A*, and the second to pretend to *A* that he is *B*. While *A* and *B* believe that they are having the exchange pictured in figure 3.2, *N* sits between them like the two-faced god Janus and what actually happens is shown in figure 3.3.

To avoid this problem, *A* and *B* need to ensure that the public keys they receive from each other are genuine. One way to do this is for each principal to disseminate a hash of her public key using a higher integrity channel. Some people, for example, myself included, list the hash of their public key (known as the key's *fingerprint*) on their business card; of course this only helps if your interlocutor receives your business card before having to send you a secret message. It doesn't work if someone you never met wishes to contact you in confidence after having



(1.1)	$A \rightarrow N : K_A;$	*A* thinks she is sending her public key to *B*. Actually, the key is intercepted by *N*.
(1.2)	$N \rightarrow B : K_\alpha;$	*N* saves away *A*'s key and sends *B* another public key he made himself, for which he has the matching private key.
(2.1)	$B \rightarrow N : K_B;$	The same thing happens ...
(2.2)	$N \rightarrow A : K_\beta;$	... in the opposite direction.
(3.1)	$A \rightarrow N : \mathrm{E}_{K_\beta}(m_A);$	*A* thinks she is sending *B* a secret message, so she encrypts it under what she believes to be *B*'s key. But this is in fact one of *N*'s fake keys, so *N* can decrypt what he gets and read m_A.
(3.2)	$N \rightarrow B : \mathrm{E}_{K_B}(m_A);$	*N* re-encrypts m_A under *B*'s genuine key and sends it along to the intended recipient, to whom everything appears as normal.
(4.1)	$B \rightarrow N : \mathrm{E}_{K_\alpha}(m_B);$	The same thing happens ...
(4.2)	$N \rightarrow A : \mathrm{E}_{K_A}(m_B);$	... in the opposite direction.

Figure 3.3. What is actually happening.

read your web page.

A high integrity channel with more widespread distribution than one's own business card is a book[25]; banking on this, a group of researchers at Cambridge, led by Anderson, produced a book, *The Global Trust Register* [20], containing a list of public key hashes[26]. Of course such a book has little hope of containing the keys of all the users in the world, but it may be helpful in bootstrapping a "web of trust", as used by Zimmermann's PGP[27] [270]. In PGP's model all the users,

[25] This is the reason why I listed the fingerprints of my keys on page xx.

[26] This had been theorized even in the original Diffie-Hellman paper [90], but doing it in practice raised a number of interesting issues. The act of publishing this book was meant by its authors to be as much a provocation as a useful service.

[27] PGP, which stands for *Pretty Good Privacy*, is a public key cryptography program that is the de facto standard for email encryption. Originally released as open source for MS-DOS in 1991, and at

as peers, mutually certify the validity of the keys of their interlocutors. Users may thus obtain unreliably certified keys over insecure channels, as long as they can build "chains of trust" starting from people they know and leading to those keys.

The "signature of Alice on Bob's key" is actually a signature on the combination of Bob's public key and Bob's name. It means: "I, Alice, solemnly certify that, to the best of my knowledge, this key and this name do match". To Charlie, who must decide on the validity of Bob's key, this statement is only worth as much as Alice's reputation as an honest and competent[28] introducer; in fact, PGP lets you assign a rating (denoted as "trust level") to each introducer, as well as a global confidence threshold that must be reached (by adding up the ratings of all the introducers) before a key can be considered as valid as one that you signed yourself[29]. For example you may require two signatures from marginally trusted introducers, or just one from a fully trusted one; but someone with a higher paranoia level might choose five and two respectively. The supremely paranoid PGP users (I am one, and I know several others) do not place any great deal of trust in the competence of external introducers and truly rely only on keys whose validity they personally verified.

In this model *The Global Trust Register* acts as a supplementary introducer, with the scientific reputations of its authors as guarantees of honesty and competence. Because it certifies a number of "important" keys (e.g. keys of crypto pioneers and activists who are likely to have signed many other keys in their respective communities), there is a non-zero chance that, with only a couple of degrees of separation, it will be possible to form chains connecting the desired key with one or more of the ones listed in the book.

---

the time the only military-grade encryption software widely available to civilian users, it was quickly ported to every imaginable platform and is now sold as a commercial product by Network Associates.

[28] These two qualities are completely independent, and both are required for the introducer to be trustworthy. A dishonest introducer might intentionally sign a falsehood such as "This key belongs to the president of ABC Bank" in preparation for a fraud, while an incompetent one might be conned into signing that same falsehood because he is insufficiently careful in his verification of the supporting credentials of the principal who requests his signature.

[29] Note also that the fact that a key is valid does not imply that it is trusted. To believe that Alice's key is valid is to believe that it really belongs to Alice. To believe that it is trustworthy (for introductions) means to believe that Alice is honest and competent at key management. It will be easy for the reader to come up with examples of cryptographically illiterate acquaintances whose keys should be taken as valid but untrusted.

3.5 Security policies

3.5.1 Setting the goals

In most engineering disciplines it is useful to clarify the requirements carefully before embarking on a project. Such advice may sound so obvious as to border on the useless, but it is of special relevance to computer security. Firstly because, as shown by Anderson [8], it is all too often ignored: diving straight into the design of crypto protocols is more fascinating for the technically minded. Secondly because security is a holistic property—a quality of the system taken as a whole—which modular decomposition is not sufficient to guarantee[30]. It is thus important to understand clearly the security properties that a system should possess, and state them explicitly at the start of its development. As with other aspects of the specification, this will be useful at all stages of the project, from design and development through to testing, validation and maintenance.

The security policy is a set of high-level documents that state precisely what goals the protection mechanisms are to achieve. It is driven by our understanding of threats, and in turn drives our system design. Typical statements in a policy relating to access control describe which subjects (e.g. users or processes) may access which objects (e.g. files or peripheral devices) and under which circumstances. It plays the same role in specifying the system's protection properties, and in evaluating whether they have been met, as the system specification does for general functionality. Indeed, a security policy may be part of a system specification, and like the specification its primary function is to communicate.

Because the term "security policy" is widely abused to mean a collection of vapid managerial platitudes, there are three more precise terms which have come into use to describe the specification of a system's protection requirements.

1. A *security policy model* is a succinct statement of the protection properties which a system, or generic type of system, must have. Its key points can typically be written down in a page or less. It is the document in which the protection goals of the system are agreed with an entire community, or with the top management of a customer. It may also be the basis of formal mathematical analysis.

2. A *security target* is a more detailed description of the protection mechanisms which a specific implementation provides, and of how they relate to a list of control objectives (some but not all of which are typically derived from the policy model).

[30] Connecting secure components together does not necessarily yield a secure system.

3. A *protection profile* is like a security target but expressed in an implementation-independent way to enable comparable evaluations across products and versions. This can involve the use of a semi-formal language, or at least of suitable security jargon. The protection profile forms the basis for testing and evaluation of a product.

3.5.2 The Bell-LaPadula security policy model

The first and most influential security policy model was proposed in 1973 by Bell and LaPadula [29]. The US military had developed a system for the classification of sensitive documents (based on the well-known hierarchy of OPEN, CONFIDENTIAL, SECRET and TOP SECRET) and rules that prevented officers with a lower clearance from reading data in documents of higher classification. Bell and LaPadula were trying to enforce this information flow policy, also known as "multilevel security", in computer systems. They boiled everything down to two rules that every object access should obey:

1. (No Read Up, or "simple security property".) No process may read data at a higher level.

2. (No Write Down, or "*-property".) No process may write data to a lower level.

The first rule, as the name suggests, is simple to understand: the cleaner is not allowed to read the documents in the general's safe. But the second rule, despite being less intuitive, is also necessary. It prevents the general from leaving his important documents on his desk (a "lower level" area compared to his safe) whence the cleaner could obtain them without violating the simple property.

This simple formalization captures the essence of "multilevel security". But the methodology itself merits attention. Once appropriately modelled, the system can be represented as a state machine. Then, starting from a secure state, and performing only state transitions allowed by the rules of the policy model, one is guaranteed to visit only secure states for the system. This allows one to derive security proofs. (Amoroso [7] gives a good description of this process, with worked examples.)

This process can be applied independently of the particular policy, as long as the policy itself is consistent. This idea of how to model a security policy formally was a significant and influential contribution from Bell and LaPadula, perhaps as important as the introduction of the BLP policy model itself.

An important concept associated with this modelling strategy is that of *trusted computing base* or TCB, which is the controlling core of the system that makes sure

that only the allowed transitions ever occur. More formally, the TCB is defined as the set of components (hardware, software, human, ...) whose correct functioning is sufficient to ensure that the security policy is enforced—or, more vividly, whose failure could cause a breach of the security policy. The goal of formalization is to make the security policy sufficiently simple, in order for the TCB to be amenable to careful verification.

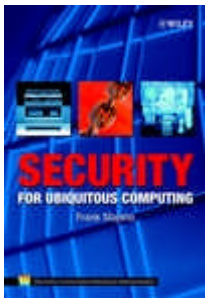
3.5.3 Beyond multilevel security

It must however be appreciated that not all security policies are sets of rules about access control. There are many contexts in which the aspect of greatest interest in the system is not access control but authentication, or delegation, or availability—or perhaps a combination of those and other properties. The Biba [33] and Jikzi [16] policies, for example, and several others that I discuss at greater length with Anderson and Lee in [17], are cases where integrity matters more than access control. These are not just a matter of controlling write access to files, as they bring in all sorts of other issues such as reliability, concurrency control and resistance to denial-of-service attacks. Policies for key management and certification, which we shall encounter in section B.3, are further examples that have no relationship to multilevel security or access control.

On a more general level, we may speak of "security policy" whenever a consistent and unambiguous specification is drawn stating the required behaviour of the system with respect to some specific security properties.

A security policy is a specification of the protection goals of a system. Many expensive failures are due to a failure to understand what the system security policy should have been. Technological protection mechanisms such as cryptography and smartcards may be more glamorous for the implementer, but technology driven designs have a nasty habit of protecting the wrong things.

There exists a spectrum of different formulations for security policies, from the more mathematically oriented models that allow one to prove theorems, to informal models expressed in natural language. All have their place. Often the less formal policies will acquire more structure once they have been developed into protection profiles or security targets and the second- and third-order consequences of the original protection goals have been discovered.



Frank Stajano (University of Cambridge)
Security for Ubiquitous Computing
John Wiley and Sons, Ltd
Wiley Series in Communications Networking & Distributed Systems
ISBN: 0-470-84493-0
Hardcover; pp. 267 (xx + 247)
Publication date: 2002-02-12
RRP: 34.95 GBP (UK); 59 EUR (rest of Europe); 60 USD (USA)

http://www-lce.eng.cam.ac.uk/~fms27/secubicomp/