Security

Security is important in all computer systems, but matters particularly on the web since there is frequent interaction with parties not previously known, and also since interactions require transfer of information on public networks where they can easily be observed or modified. Standard security techniques, such as firewalls, are designed to separate the good from the bad, but in the case of the web we actually wish to open up a window in this wall – after all there is no use putting up a web server that no one can talk to. Recall the requirement for "limited trust" on the internet. We cannot keep every stranger out just to keep the "bad guys" out. So security gets more complicated.

1 Threats

There are three main properties that a secure information system should satisfy.

1. Privacy/confidentiality. Make sure that private information is not made accessible to an unauthorized party. E.g., a web server should not let one customer see the purchase history of another customer.

2. Data integrity. Make sure that unauthorized changes cannot be made to the data. E.g. a web server should not let a customer change their account information to show a past due account as paid-in-full.

3. Service integrity. Two main ways to lose service integrity, besides accidents and acts of nature, are:

   - Vandalism. This is the sort of havoc wreaked by many viruses. There may be no direct benefit to the attacker, unlike in the first two cases. But the loss suffered by the victim can be great.
   - Resource Stealing. E.g. in a denial-of-service attack, a bogus "client" site can send out thousands of fake http requests each second, completely overwhelming the server and not leaving it any cycles to serve genuine requests.

Resource stealing attacks typically take the form of "denial-of-service". They are very hard to fight without hurting legitimate traffic. For instance, most web-sites will limit the number of requests they handle from one IP address in any period. To counter this, most attackers will spoof IP addresses, and pretend to have requests coming from many different machines. Now web-sites have to look at what is being requested to determine whether the requests are legitimate or not. This can become expensive to do in itself. Unwanted e-mail "SPAM" also has many characteristics similar to resource-stealing attacks on the web.

Vandalism attacks are typically due to viruses that have been transported as part of code. This code could be in "helper" applications supporting web transfers of some data types. With dynamic web pages, we are often moving executables along with data, so care must be exercised.

Focusing on the first two properties, we find that they could be impacted both on a web client and on a web server. A few common communication-related attack techniques are:

- Masquerade – pretend to be someone else. Appropriate authentication can be used to fix this.

- Address Spoofing – pretending to have a different IP address than actual. This is one way to masquerade.

- Eavesdropping – listen in to communication between client and server. Can exted to *data diddling* where the communication is not just listened to but also modified. Encryption of data transmission is used to fix this problem.

- Man-in-the-middle – a relay point pretends to be the server to the client and the client to the server. Permits regular sessions to be established at first, between it and both parties, and then exploits this for eavesdropping and data diddling.

- Replay Attack – An eavesdropper can record a communication session, and then replay (appropriate portions of) it to masquerade as one of the parties in the communication. E.g. if you telnet to a remote machine and transmit your username and password over an unencrypted channel, someone can record this, and login at a later time pretending to be you.

Our primary concern is to protect our selves as we transact business with strangers. We consider defense mechanisms necessary for the ordinary course of business in the bulk of this chapter. We will close with a brief section on additional protections against explicit non-transactional attacks on a web servevr, most of which apply to any networked computer.

2 Defense Mechanisms

Authentication: The first requirement in security is to make sure that you know who you are dealing with. *Authentication* is the process by which the identity is established of a party to a transaction. Typically, authentication is performed by means of credentials of some sort. An ID badge frequently serves this purpose in the physical world. A password is common in the electronic world.

Note that in a typical transaction, there are at least two parties involved, and each must satisfy themselves that they are dealing with who they think they are dealing with. In many instances, the parties to a transaction are asymmetric in that one is big and well-known whereas the other is small and unknown. E.g. when you go to physical store to make a purchase, you "know" the merchant with whom you are entering into a purchase transaction. But the merchant does not know you, and may require to see your driver's license, obtain your signature, and so on, as authentication means. That is to say, authentication only takes place one way. On the web, there still may be asymmetry, but one is frequently dealing with small merchants or other individuals so that all parties to the transaction need to authenticate. Even when a large, well-known merchant is involved, authentication is still required because of the ease with which an interloper could be masquerading as the merchant. For example, you may think you are at your bank's web site when you are really at the site of an attacker pretending to be your bank. See Fig. 1. In other words, authentication is required in each direction. We will discuss below how that is accomplished.

Non-repudiation is a property of a secure system whereby a party to an action cannot in the future deny that it performed this action. In the physical world, signatures are frequently used for this purpose. In the electronic world, corresponding digital signatures have to be used. Suppose you log on to your bank's web site and withdraw some money. Could you, the next day, claim that it was not you, but someone else, who did this? The answer is "Yes!" Mere log in is insufficient for non-repudiation. For example, the bank's transaction recording system may be buggy (or been hacked into, or even intentionally set up to defraud customers). If you challenged the bank, neither side would have irrefutable evidence. Who wins in court depends on circumstantial evidence, including such "extraneous" considerations as which account the withdrawn money was transferred to.

Authorization: Once we know who, the next question is what they are allowed to do. Authorization is the granting of privileges to an (authenticated) party to a transaction to perform certain actions. Typical actions are read and write. In general, an authorization can be viewed as a triple comprising a *subject* having the privileges to perform an *action* on an *object.* A subject with read privileges but no write privileges on a particular data resource cannot modify the data, though it can read it. In a large system, there will be many data resources (objects) and many users (subjects); so specification of authorization by listing all the individual authorization triples is not practical. Instead, authorization policies are typically specified in terms of classes of users having specified privileges on classes of resources. For example, we may permit all faculty in a department to view the grades of any student in the department, but only selected faculty (such as a course instructor) may be permitted to modify these grades.

Policy: A security *policy* is a specification of the authorization rules that the system is expected to follow. This is a statement of what the end result should be, and not a specification of how to go about achieving it.

Mechanism: A security *mechanism* is the means by which the desired security policy is implemented. There are many possible components of a security mechanism. Perhaps the most fundamental one is *Encryption.* So we shall study it in some detail next. Note that encryption by itself is a tool and not an end-goal. If you think of security as a shield, then encryption is a tool with which you can make a shield, but it is not itself a shield.

3 Encryption

Encryption is the act of computing a reversible function of some data. Decryption is the inverse, recovering the data from the encrypted form. Each of these functions is carried out based upon an algorithm (such as RSA, DES, PGP, Caesar, etc) and a key. The algorithm is assumed known. The key is (usually) secret. Encrypted data cannot be interpreted by some one without the decryption key, and so encryption is at the heart of most schemes to prevent unauthorized access.

3.1 Symmetric Key Encryption

Given a *plaintext* string s, it is encrypted, using an encryption key, K_{enc}, to obtain a *cyphertext* string $K_{enc}(s)$. Decryption using a matching decryption key K_{dec} permits the recovery of the original plaintext string: $K_{dec}(K_{enc}(s)) = s$. See Fig. 2.

Encryption of secret messages has been used since early human history. Julius Caesar is said to have used the so-called Caesar Cipher to communicate with his generals, in which each letter is substituted by a letter three positions later in the alphabet. Thus each A is replaced with a D, each B with an E, and so on. See Fig. 3. Augustus Caesar is said to have preferred a cipher with a shift of one position.

More generally, one can consider substitution ciphers, in which each occurrence of a letter in the plain text is replaced by a substitute letter in the cipher text, using a substitution table.

To make substitution ciphers harder to break, one technique that could be used is to substitute larger chunks of text – not just single letters. This family of ciphers is called *polygram ciphers*. Unfortunately, the size of the substitution table grows exponentially with the number of characters in the substitution unit. See Fig. 4. Suppose we have 100 distinct characters in the alphabet. A single character substitution table would have a 100 entries, a two character table would have 10,000 entries, a three character table would have 1,000,000 entries, and so on. A six character polygram cipher would require a substitution table with a trillion entries!! Clearly, this is not practical.

To limit this growth, it is common not to use a stored table, but rather to have a substitution rule, in effect a means for deriving the necessary rows of the table, based on a small encryption key. How large should the encryption key be? That depends on how fast the computers are that are trying to break the key. Consider a trivial key comprising a single bit. This bit can either be a 0 or a 1. We can check both possibilities – one of them must work. As the length of the key increases, the number of possibilities to check grows exponentially. With a b bit key, we have 2^b possible keys to consider. Suppose that a current day computer runs at 3 GHz and requires 300 instructions to check a possible key. In one second, this computer can run through 10 million possible keys. In a day, it can do almost one trillion. That is approximately 2^{40}. In other words, a simple PC today can break a 40-bit key in about a day. But with a 60-bit key, it will take a 100 CPU cluster running for 3 years.

The above rough calculation used a very simple-minded enumeration technique to guess a key. If there are observable patterns in the encrypted information, we can do much better than this. For example, consider a simple single character substitution cipher used to encrypt English language text. If we guess that the most frequently occurring character in the encrypted string corresponds to the letter "E" in the English alphabet, we have a decent chance of being right. Certainly, guessing popular characters is a much smarter strategy then enumerating all of them in random order.

In general, breaking a cipher involves finding the decryption key. There are two primary ways to try to do this. The first is through the mathematics of factorization. Knowing the encryption technique, one knows what the decryption key should look like and can start trying various combinations. It is easy to count the number of different possibilities, and hence the number of CPU cycles spent. The second way is by frequency analysis – looking at the cipher text, one can exploit statistical properties of the observed text to make guesses at the original. (This is easy to understand for a simple substitution cipher. Given a piece of cipher-text, you could guess that the most frequently occurring character corresponds to the letter "e" and so on. The techniques for more complex ciphers are similar in spirit, though far more sophisticated in terms of the mathematics.) The benefits one can obtain from frequency analysis depend quite a bit on having enough encrypted text around to be able to derive meaningful statistics.

To prevent the code-breaker from getting any such hints, a good encryption scheme should produce cipher text that looks as close to "random" as possible, no matter what patterns there may be in the plain text input. One requirement for a string to be random is that each character should occur approximately equally often. (This is a necessary, but not a sufficient condition – consider a string comprising all the letters of the alphabet in order, "abcdefghijklmnopqrstuvwxyz." This string has each character appear an equal number of times (exactly once) but it is certainly not random.) If we use binary notation, this randomness becomes easier to think about. Let us first convert the input plain text into a sequence of 0s and 1s. This is easy to do, since a computer internally represents each character that way. Now create a substitution rule that involves computing the exclusive OR of the input string with a key string that comprises approximately equal numbers of 0s and 1s. The exclusive OR will flip the bits in the input string that correspond to a 1 in the key string, and leave unchanged the bits that correspond to 0. In this way, approximately half of the input bits will be flipped, and the output will have approximately equal numbers of 0s and 1s no matter whether the input had 0 or 1 predominate. See Fig. 5.

The general ideas described above have been embodied in a very popular encryption mechanism known as *DES* (Data Encryption Standard). This uses a 56-bit key to encrypt data, one 64-bit block at a time. DES encryption is performed in 16 rounds. In each round an exclusive OR is computed of the data with a function of the key. Of course, computing this exclusive OR repeatedly with the exact same blocks of bits is silly. Rather, each round also involves a permutation of bits in both the data and the key, using standard permutation tables. See Fig. 6.

The 56-bit DES scheme has been shown not to be secure enough. With exponential growth in CPU speeds, it has become possible to crack a 56-bit DES cipher within a fairly short period of time. To address this problem, the triple-DES scheme has been proposed, with a 168-bit key. This is simply the basic DES scheme applied thrice, each time with a different 56-bit fragment of the 168-bit key. See Fig. 7.

When transmitting DES encrypted data, the transmission proceeds in independent 64 bit blocks, each of which can be coded and decoded independently. See Fig. 8.

An alternative to dividing the data into blocks is to have a way of generating a very long key. For instance, a popular book can serve as a long key – each character in the message can be encoded using one character from the book: each character occurrence in the message is thus encoded differently, as long as the book is longer than the message. Electronically, the goal is to generate a potentially infinite pseudo-random bit-stream with which the message can be XORed. Such a scheme is called a *stream cipher*, credited to John Vernam of AT&T Bell Labs. A popular family of such ciphers are Ron's Ciphers, named for their inventor Ronald Rivest. RC4, in particular, is very widely used, including in TLS/SSL, which we discuss below. The basic mechanism is to take a 128-bit key and perform permutation, shift and XOR operations on it to generate the required pseudo-random keystream. The plain text can then by XORed with this keystream. See Fig. 9.

In traditional encryption schemes, such as those described above, the same key is used to encrypt and decrypt. This is why such schemes are collectively called *symmetric key encryption* schemes. The key is a shared secret between the sender and the recipient. In the case of a substitution cipher, this key is the substitution table. In the case of an exclusive OR based encryption, it is simply the encryption bit string. Bit operations can be performed very efficiently in hardware, and hence encryption schemes based on bit manipulation (permutations and exclusive OR) can be provide fast encryption and decryption. In the case of DES, as for many other symmetric key schemes, the encryption and decryption algorithms are identical.

Some secure means has to be found to share the key before it can be used in a symmetric key encryption scheme. You may have seen war movies where rival militaries invest huge amounts of effort to try to determine the the other sides "codebook" (or set of keys). Getting the key securely to the parties involved in a transaction is not easy, and this is known as the *key distribution problem.* Once you have gone to the trouble of establishing a shared key, you would like to use it for a while. Yet, for a variety of reasons, you may need to stop. For instance, if one of the parties "in" on the secret is to be removed from the group, we have a key revocation problem. Unfortunately, the only way to accomplish this is to stop using the current key, in effect revoking it from everyone, and then set up a new shared key among the parties still in the group.

3.2 Assymetric key encryption

Assymetric key encryption is used to address the key distribution problem (and additionally gain several other benefits). Public-private asymmetric key encryption involves a pair of keys, such that one can be used to decrypt messages encrypted by the other. One is called the public key, because it can be made known to everyone, whereas the other is called the private key and is kept completely private – never shared with any one. A sender s encrypts a message m with its private key to obtain $priv_s(m)$. A recipient may decrypt this using the matching public key pub_s, to obtain $pub_s(priv_s(m)) = m$.

Thus the key distribution problem is solved. If a sender s wishes message m to be read only by recipient r, the sender encrypts the message with r's public key, and sends $pub_r(m)$. Only r is able to decode this message, using its private key, known to no one else. $priv_r(pub_r(m)) = m$.

One other benefit of asymmetric keys is that they can be used as a signature mechanism. If s wishes to sign message m, it simply computes $priv_s(m)$. Anyone can decrypt this message, using pub_s, which is known to everyone, as discussed above. But in doing so, they will authenticate the message m as having been encrypted (that is, signed) by s. See Fig. 10.

This sort of public key cryptography was first proposed by Diffie and Hellman in 1976, but the most popular algorithm today is due to Rivest, Shamir and Adelman [1978] and is know as the RSA algorithm using the first letter of the last names of each of the three inventors. (Rivest, Shamir, and Adelman won the Turing Award, the highest prize a computer scientist can win, for their work on RSA, in 2003. Ron Rivest is also the creator of the RC4 and MD5 schemes, among others.)

The functions used for encryption and decryption are very different from those for symmetric key methods. The mathematics is non-trivial. Here we will give you a flavor of how it works. The central idea is that factorization of large numbers is hard. In particular, suppose that you define a number n, obtained as the product of two prime numbers p and q. Given n, it is very hard to find its factors p and q. However, given p and q, it is easy to compute the product n. To take this idea and develop an encryption scheme from it, we need two results from number theory, which we describe next.

The first result is *Fermat's Little Theorem*, which says that for any prime number p, and any natural number a, $a^p = a \ (mod\, p)$. Equivalently, unless a is a multiple of p, $a^{p-1} = 1 \ (mod\ p)$. Perhaps you have not see the "mod" notation before. The idea is that instead of arranging numbers on a number line, we arrange them in a circle, and just keep going around the circle forever, clockwise for larger numbers and counter-clockwise for smaller numbers. The size of the circle is called the *modulus* and arithmetic on this circle is called *modulo arithmetic*. If we have modulus m, then there are m distinct "spots" on the circle, and integers that are m apart will fall on the same spot. That is, if $a = b + km$, for some integers a, b, k, then a and b will be considered equal $(mod\ m)$. See Fig. 11. One simple way to perform modulo arithmetic is to divide both sides of the equation by the modulus and compare the remainders: if these are equal, the equation holds. Thus, Fermat's Little Theorem says, in English, that if you raise any natural number to the power of a prime number, then it is equal to itself, modulo the prime.

Example 1 *Consider a choice of* $a = 4$. *When* p *is chosen to be* 3, *we get*
$a^p = 4^3 = 64 = 1(mod\ 3) = a(mod\ p)$ *and*
$a^{p-1} = 4^{3-1} = 16 = 1(mod\ 3) = 1(mod\ p)$. *With the same* a, *if we choose* p *to be* 5, *we get*
$a^p = 4^5 = 1024 = 4(mod\ 5) = a(mod\ p)$ *and*
$a^{p-1} = 4^{5-1} = 256 = 1(mod\, 5) = 1(mod\, p)$. *Similarly, choosing* p *to be* 7 *and* 11 *respectively, we get* $a^{p-1} = 4^{7-1} = 4096 = 1(mod\, 7) = 1(mod\, p)$ *and* $a^{p-1} = 4^{11-1} = 1048576 = 1(mod\ 11) = 1(mod\ p)$. *However, if we choose* p *to be* 2, *then* a *is a multiple of* p, *so we get*
$a^p = 4^2 = 16 = 0(mod\ 2) = 4(mod\ 2) = a(mod\ p)$ *but*
$a^{p-1} = 4^{2-1} = 4 = 0(mod\ 2) \neq 1(mod\ p)$.

The second result is the *Chinese Remainder Theorem*, which considers a set of congruences of the form $x = a_i \ (mod\ p_i)$ for $i = 1, ..., k$. The theorem states that there exists a solution for x if each of the p_i is relatively prime to every other $p_j, j \neq i$, and provides a constructive procedure to find this solution. Furthermore, this solution is found modulo the product $\Pi_{i=1}^{k} p_i$. In the special case where $k = 2$ and $a_1 = a_2 = 1$, we have that the solution is $x = 1 \ (mod\ p_1p_2)$.

Example 2 *Given* $x = 1 \ (mod\ 4)$ *and* $x = 1 \ (mod\ 5)$, *the Chinese Remainder Theorem says that the solution is* $x = 1 \ (mod\ 20)$. *We can check that this is the case by choosing x to be 21, for example. This gives a* *remainder of 1 upon dividing by any of 4, 5, or 20, and hence is equal to 1 modulo each of those numbers.*

Now back to the algorithm, we have chosen two large prime numbers p and q, with a product $n = pq$. We now choose a number e, which must be smaller than λ, the product of $p-1$ and $q-1$.[1] Find another number d such that $de = 1(mod\ \lambda)$. Now n and e together serve as the public key. d, inconjunction with n, is the private key.

To encrypt a message m, we compute $m^e\ (mod\ n)$, and obtain its ciphertext c. To decrypt c, we compute $c^d\ (mod\ n)$. To see that this gives us back the original message m, compute $c^d = m^{de} = m^{k\lambda+1}$, for some integer k. This last equality flows from the way we chose d so that $de = 1\ (mod\ \lambda)$. Remembering how λ is defined, and performing some algebraic manipulations, we get that this number is $m * (m^{(p-1)(q-1)})^k$. Now recalling Fermat's little theorem, we have $m^{p-1} = 1\ (mod\ p)$ and $m^{q-1} = 1\ (mod\ q)$. Putting these two together with the Chinese Remainder Theorem, we get $m^{(p-1)(q-1)} = 1\ (mod\ n)$, where $n = pq$. Therefore, we can, modulo n, compute $c^d = m * (m^{(p-1)(q-1)})^k = m * (1)^k\ (mod\ n) = m\ (mod\ n)$.

Example 3 *Suppose we choose 7 and 13 as our two prime numbers p and q. (In practice much larger numbers would be used – we choose small primes here to demonstrate the idea). Their product is $n = 91$ and $\lambda = 84$. If we choose $d = 5$ and $e = 17$, then we have $de = 1(mod\ \lambda)$. This gives a public key of 91,17 and a private key of 5 (in conjunction with 91).*

Let us take a very small message for this example. Say $m = 2$. Then the public-key encrypted message is
$m^e(mod\ n) = 2^{17}\ (mod\ 91) = 131072\ (mod\ 91) = 32.$
To decrypt this, we compute
$32^5\ (mod\ 91) = 33554432\ (mod\ 91) = 2\ (mod\ 91) = m.$

Conversely, we can use the private key to sign the message, by computing $m^d(mod\ n) = 2^5(mod\ 91) = 32$, which just so happens to be the same as the public-key encrypted message in this case, just by chance. We decrypt this using the public key as
$32^{17}\ (mod\ 91) = 386856262276681335905976 32 = 2\ (mod\ 91) = m.$

Computing large exponents of natural numbers accurately is very challenging – note that the answer is required to be accurate to the last digit – we

[1]Variations of the algorithm use the least common multiple rather than the product. They may also compute something called the *totient* function, which we will not discuss here.

cannot use approximations. We exploit the fact that we are not interested in the full answer, just in the answer modulo n. A standard technique to compute the requisite function is in a first phase to determine, iteratively, the value of m^2 $(mod\ n)$, m^4 $(mod\ n)$, m^8 $(mod\ n)$, etc. At each stage, taking the modulus keeps the size of the answer down to no greater than n. The computation is simply a squaring (product with itself) of the previous number, and then division by n. In a second phase, the bit positions with a value of 1 in the exponent (when expressed in binary notation) are used to select elements in the sequence of powers computed in the first phase, and the product of these selected elements computed.

Example 4 *Suppose we wish to compute* $2^{693}(mod\ 5)$*. We could first compute* 2^{693}*, which is 4109481173084666802532023346000100519961202970955 6045777330319555224469955445943922763019814668659775210804444188889 2325882964314454560967680686052895717819140275184930690973423372373 1084712712286819785291857792. Then we take* $(mod\ 5)$ *of this huge number, to get* 2 *as the final answer. Note that* 2^{693} *requires about* 694 *bits to represent completely in a computer. Most computers today are designed to limit their representation of numbers to* 32 *or* 64 *bits. So special logic is required to manage this large number, in pieces.*

In contrast, consider the powers technique, shown in the table below:

Power p	*1*	*2*	*4*	*8*	*16*	*32*	*64*	*128*	*256*	*512*
$2^p(mod\ 5)$	*2*	*4*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
693 in binary	*1*	*0*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*1*
$2^{693}(mod\ 5)$	*2+*	*0+*	*1*	*0+*	*1+*	*1+*	*0+*	*1+*	*0+*	*1*

We begin by computing powers of 2. $2^1 = 2$, $2^2 = 4$, $2^4 = (2^2)^2 = 4^2 = 16 = 1(mod\ 5)$, $2^8 = (2^4)^2 = 1^2(mod\ 5) = 1(mod\ 5)$, $2^{16} = (2^8)^2 = 1^2(mod\ 5) = 1(mod\ 5)$*, and so on, to get the second row of the table. We then express* 693 *in binary to select the powers of* 2 *that add up to make* 693 *in the third row of the table. (Note that the binary representation gets written backwards in the table, because the powers of* 2 *are increasing from left to right). Finally, we must add up the powers indicated by the 1s in this row. To this end, in the fourth row, we reproduce the entries from the second row where the corresponding third row entry is* 1*. Adding these up, we get 7, which is equal to* $2(mod\ 5)$*. Notice that we only dealt with small numbers throughout, using this technique, rather than the very large numbers required for a literal evaluation.*

The major drawback of asymmetric key algorithms is that they are computationally intensive, even with the more efficient computation techniques described above. As such, it is typical not to encrypt an entire message for signature purposes, but rather to encrypt only a digest of it. Given a large message m, we compute a small digest of it $h(m)$. The sender then signs only $h(m)$ and not m. The recipient receives m, and recomputes its digest $h(m)$. It can then compare this to the signed digest received. If the two match, it is extremely unlikely that the message has been tampered with.

An evil-doer can get away with changing m as long as that change does not affect $h(m)$. For example, suppose we define $h(m)$ as a simple exclusive OR of m. (In other words, a parity bit computation). Now an attacker can make the desired critical changes to m, such as the value of a contract, or the account number to which funds are to be transferred, and compensate with additional non-critical changes (e.g. in the greeting line or other surrounding text), so that the parity computation remains unchanged. How do we make sure that this sort of attack is not possible? This is accomplished by using a *one-way function* to compute the digest. A one-way function has the property that it is easy to compute, but its inverse is virtually impossible to compute. We wish to define a one-way function $h()$ such that, given a message m of arbitrary size, the result $h(m)$ will have a fixed small number of bits, so that many different messages will have the same digest. However, the property of the one-way function $h()$ makes it very hard to find this class of messages.

A popular algorithm to compute a message digest is known as *MD5* [Rivest]. This algorithm pads the given message with one 1 bit, zero or more 0 bits, and a 64-bit value for the original message length in bits, to obtain a bit sequence that is a multiple of 512 bits long. It then performs a sequence of bit operations on the sequence of 512 bit blocks representing the message to compute a digest that is exactly 128 bits long. See Fig. 12.

To recap, a signed communication with asymmetric key encryption proceeds as follows: Starting with a message m, the sender computes a digest $h(m)$, and signs the digest $priv_s(h(m))$. It appends this to the original message and sends it. The recipient independently computes $h(m)$ from the former and compares this with the $h(m)$ obtained by computing $pub_s(priv_s(h(m)))$. If these two are the same, the recipient knows that it received the message that s wanted to send it and that it has not been tampered with. See Fig. 13.

Signatures are often dated in real life? How should a digital signature be dated? One simple way is to include the date in the body of the message itself. A more common way is to sign a concatenation of the message digest and date. That is, the sender communicates $(priv_s(date, h(m)), m)$. The recipient can thus see the date of the signature when it decrypts the signature in the standard way, using pub_s.

In practice, how are public/private key pairs managed? First, they must be generated, using a method simlar to what we saw above. Let us assume that each person has the technical competence to do this. (If not, they have to get some trusted competent party to do this for them). The public key can now be published. (E.g. PGP public keys you may have seen from some of your e-mail correspondents). The private key must be kept in some safe place.

But we have a few worries. These keys are not small, and they are not mnemonic – they look like a random sequence of characters. So they are hard to remember, unlike a typical password that you create for yourself. So they are likely to be stored digitally. But this raises additional concerns. This digital storage had better be secure – typically password protection is applied to it, in addition to physical security. We have to concerned not only about theft, but also of loss. What if your private key is on a disk drive that crashes? How do you regain your identity? One solution is that you should use a *key escrow* as a backup. You store a copy of the key with a trusted party, possibly password-protected. The trusted party promises not to look at your key, and just to get it back for you should you need it in the future.

No security procedure can be completely fool proof. What if a private key is stolen? There needs to be a mechanism to revoke this key and not have it apply in any future transactions. The only way to accomplish this is through changing the associated public key that has been published, and potentially cached at a number of places. Since you have no control over who has squirreled away a copy of your public key, you cannot force a cache flush and reload. To deal with this eventuality, published keys often will have expiration times associated with them. By this means, if the secret is compromised, there is only a finite time window during which there is the potential of exposure.

4 Secure Web Connection

Secure web connection protocols bring together the mechanisms described above, and are described here. To defend against the attacks above, the tools used are (i) Authentication of client and server, and (ii) Encryption of transmitted messages. We have seen how encryption works in the preceding section. For authentication, digitally signed *certificates* can be used, as we will shortly see.

Secure communication over the web is most often accomplished using the *HTTPS* protocol, which is simply the http protocol run on top of *TLS* (transport layer security), or its predecessor, *SSL* (secure socket layer). The security layer runs above the transport layer (TCP) but below the application layer (HTTP). Even though it is called “transport layer security” it is really independent of the transport layer protocol. In fact, it is commonly implemented at the application level, and indeed the HTTPS “protocol” is HTTP running on top of TLS. By calling this an entirely new protocol, and assigning a different default port number to it, we make it possible for the same machine to support both secure and insecure HTTP requests, using different ports.

The basic principle in TLS is to work in two phases. In the first phase, the parameters of a secure session are negotiated, and a session key is established and shared. In the second phase, transmissions are encrypted using the shared session key and a symmetric key encryption algorithm, such as DES or RC4. The first phase is run using public-private key pairs and the RSA algorithms. It begins with the server authenticating itself to the client. Once the client is satisfied, it can communicate its public key, and encryption capabilities, back to the server. Now the two can establish a shared session key and transition to the second phase. Note that all actual requests and responses are encrypted using the shared session key. See Fig. 14.

The two-stage process has a number of benefits. First, it uses the more efficient symmetric key protocol for the bulk of the communication. The first stage of the protocol is used to establish the shared key, and thus address the difficult key distribution problem. A second benefit of using a session key established as above is that only a small amount of data is ever encrypted using any one key, giving a code-breaker very little information with which to work. A third benefit is that the shared secret is shared only for the duration of the session – thus we do not have issues of revocation to deal with. This is particularly important on the web where one may wish to transact on a one time basis with parties who are not well-known to us and with whom we do not expect to maintain a continuing relationship. Finally, the first phase of the protocol performs crucial server authentication.

How can a server authenticate itself to the client? It would appear that sending something encrypted with the server’s private key would suffice. A client can decrypt this using the server’s public key, and know that it has received a valid message from the server. There is a problem with this basic scheme. How does the client know the server’s public key? There are thousands of servers, and it is unreasonable to expect that a client will know public key information for all of them. The client could ask the server to tell it its public key, but in that case an interloper could send it a false public key and hence be able to access messages intended for the server from the client. The solution is to use indirection – have a client know about a "trusted directory of public keys" (or a few of these), and have these know the public keys of various servers. These solutions are put together in the form of a *certificate authority.* There are only a few of these, and one can expect that clients will know most of them. A server obtains a *certificate* from the certificate authority and presents this to the client. This certificate is signed by the certificate authority, and this signature can be verified by the client if it knows the public key of the certificate authority. The certificate has in it the server's name, public key, and other relevant information. Note that the certificate does not say anything about the IP address or even who the web server is. Indeed, it is possible for an evil-doer to steal a certificate and present it in an attempt at masquerade. The reason we do not have to worry about it is that the certificate includes the public key of the intended server. Any message encrypted by this key can only be read by the holder of the corresponding private key: of course, the evil-doer does not have access to this private key.

A server presents this certificate to the client at the start of a session. Notice that the certificate authority is not directly in the picture at this stage – it signed the certificate ahead of time. This is critical since there could be millions of sessions taking place simultaneously, and there are only a few certificate authorities, so keeping them out of the critical loop is essential if they are not to become a bottleneck. See Fig. 15.

If the client has a certificate, it presents it to the server at this point. Most clients do not have certificates to present. So the client is not authenticated to the server at the time that the secure connection is established. All the server knows is that "someone" is talking to it. (At the application level, the server would typically seek additional authentication, such as password, as the first thing once a secure session was established. Notice that this client password does not go to the server "in the clear" – it is encrypted by the session key and hence not usable by an eavesdropper.) If the server requires that the client be authenticated, then the attempted connection fails at this point if the client is not able to present a satisfactory certificate.

Now the client generates a "pre-master" key for the session, and sends this to the server, encrypted by the server's public key. Since only the server can decrypt this message, no one else can get access to the this shared pre-master key. Using this pre-master, the client and server generate a master key independently, using the same algorithm, and from the master, a pair of session keys, one for communication in each direction.

5 Web server security

When you build a web server and deploy a web site, you do so because you would like to attract visitors to your site. Many of these visitors will not be known and trusted parties. Some may wish you ill. Yet, if you construct a dynamic web site with server side programs, you are letting anonymous users invoke these programs. In this section, we touch upon a few things to keep in mind when doing this. The issues here are different than for communications security discussed thus far – here we are worried about malicious programs (and data) rather than eavesdropping and masquerade.

First, whenever there is user input expected, whether in form fields, URL search strings, HTTP header fields, or any other place, do not assume that the user is doing what you expect the user to do. Even with cookies (we will study cookies in detail in the next chapter), which are created by you, make sure that the user has not tampered with them in sending them back to you. With inputs, there frequently are special characters that get handled differently by some applications. (Recall, for example, various special characters in HTML. To name one metacharacter that could be problematic, the back-tick (single open quote) character can be used to call system functions from Perl). To prevent such problems, it is good practice to disallow any characters in the input that could potentially cause harm. A simple function can filter the input to permit only the allowed characters, alphabets, numerals, and selected punctuations. Occasionally, there may be a legitimate need to use one of the disallowed characters. To support this, your filter function should make sure that the character in question is "escaped" properly, using the escape sequence requirements of the language in question.

Buffer overflows are the basis for many attacks. You are likely to have fixed limits on sizes for many inputs. If the input provided exceeds this size, make sure to check for this, and to handle it in some graceful manner – either truncating the input or going back to the user and requesting a smaller input. What you do not want to have happen is for the large input to cause some buffer to overflow somewhere else in your code, potentially permitting an attacker to overwrite locations of address space beyond just the input buffer.

It is also good not to make available to visitors information about your site that you do not want them to have. E.g. names of files are visible as part of the URL. Make sure that the name of an executable does not reveal too much about it. For instance, if you are using a popular program, rename the executable. Where feasible, even modify file name extensions so that a visitor does not know which type of software you are running on the server side.

Finally, if you are using multiple technologies on the server side, think carefully about how they may interact with respect to error situations. If one directory has files that could be written (or modified) by a program that accepts visitor input, then make sure that no file from this directory may permit server side includes. In other words, even if a visitor is able to manipulate what is written into a file, this file should be guaranteed to be in a semi-isolated area, so that it cannot in turn read some other file with sensitive data, even if it is directly requested as a resource.