Cryptography

December 12, 2016

1 Symmetric Key Cryptography

Symmetric key cryptography is the oldest and most classical notion of cryptography, and involves two parties using pre-determined information to encode a message so that both parties are able to distinguish the meaning of the message, but anyone without the pre-determined information cannot. The pre-determined information is often called a *key*, or more specifically, a *symmetric key*, since both parties have it. Such a framework provides *confidentiality* - no one besides the two parties privy to the key can read the message.

More formally, if Alice and Bob both agree upon a key K, then Alice can *encrypt* a message with the key K and send the encrypted message over to Bob, who can decrypt the message to read its contents. The central idea behind encryption is that we want a mathematically sound way of transforming the raw message, called the *plaintext*, into a pseudorandom, impenetrable string, called the *ciphertext*, which can then be unscrambled by re-applying K. Often times if the encryption algorithm is known ahead of time, an adversary can simply figure out K through brute force, by putting every possible permutation of bits into the encryption algorithm until one result matches the encrypted message. Thus, encryption algorithms should have the property that finding the key through brute force takes prohibitively long with modern computational power and algorithmic techniques. This alludes to a more central, intuitive, yet powerful cryptographic principle.

Kerkhoff's principle: *A cryptosystem's security should depend only on the secret key, and should remain secure even when adversaries are privy to any and all information about the cryptosystem, except the secret key.*

Given that an eavesdropper Eve might have any level of access to or information about the cryptosystem (except the key), let's define some potential attack scenarios to be mindful of.

1. Ciphertext-only attack: The standard example of a security threat. Eve possesses a single encrypted message.
2. Known plaintext attack: Eve possesses a single encrypted message, but also has some information about the plaintext.
3. Chosen-plaintext attack: Eve has access to the ciphertexts of arbitrary plaintexts of her choice, e.g. by tricking Alice into encrypting messages sent by Eve under false pretenses.
4. Chosen-ciphertext attack: The inverse of the chosen-plaintext attack. Eve has access to the plaintexts of arbitrary ciphertexts of her choice.
5. Chosen-plaintext/ciphertext attack: The union of the chosen-plaintext and chosen-ciphertext attacks. Eve has access to plaintexts of ciphertexts of her choice as well as the ciphertexts of plaintexts of her choice.

Under any of the above scenarios, our encryption scheme shouldn't provide Eve with any information about the message that she wouldn't already have in a ciphertext-only attack. This is equivalent to our cryptosystem remaining secure even under chosen-plaintext/ciphertext attacks, which are the most serious threats and include the other four scenarios.

One-time pad: An encryption scheme in which messages are converted to bitstrings, the symmetric key is a random bitstring of equal length, and the encryption algorithm consists merely of computing the exclusive-or of the message with the key (this has the interesting property of being its own decryption algorithm). This is a special encryption scheme because, assuming the key is selected at random, it is mathematically proven to be unbreakable, in the sense that under any attack scenario, an attacker can't do computationally better than brute force.
The drawback to the one-time pad is that the secret key must be re-selected every time a message is to be encrypted, since if symmetric keys are reused and Eve has access to two ciphertexts $c_1 = m_1 \oplus k, c_2 = m_2 \oplus k$, for plaintexts m_1, m_2, then she can compute $c_1 \oplus c_2 = m_1 \oplus m_2$, which provides her with some information about the messages which she obviously wouldn't have had otherwise.

2 Block Ciphers

The fundamental building blocks of symmetric encryption algorithms are block ciphers, which are computable functions $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. In other words, block ciphers are algorithms which are dependent on a key K and bijectively permute the bits of an n-bit bitstring. Denoting E_K as the block cipher with key K, we obtain a bijection from the set of n-bit bitstrings to itself, which is guaranteed to have an inverse D_K which we can use as the decryption algorithm. Because block ciphers operate on messages of fixed length, variable-length messages are separated into blocks which are then passed through the block cipher and then combined in some invertible way, such as concatenation.

As with any encryption scheme, block ciphers are not immune to brute force attacks which try to determine the key by trying every possibility, but with randomly selected keys and block ciphers which operate psuedo-randomly on the key, there are 2^k possible value for a key of length k to take, a search space which quickly becomes prohibitively large for relatively small values of k. The underlying assumption here is that block ciphers should behave like random permutations, in the information-theoretic sense that given a black box which randomly permutes bits and a black box which executes a block cipher, no information is gained by executing either black box any number of times on arbitrary input.

When building encryption algorithms out of block ciphers, we break the message down into blocks of appropriate length which can be passed through the block cipher. However, another concern must be addressed - if two portions of the message happen to contain the same bits, then the block cipher will map them to the same ciphertexts, so that an eavesdropper who knows the block cipher being used could determine which portions of the message are the same, allowing her to gain more information than she would have otherwise. To prevent this, we can introduce elements to the block cipher which are different for every block, as the following common scheme demonstrates.

Cipher block chaining: Whereas a naive encryption scheme might use a block cipher E_K by encrypting each block M_i and concatenating the ciphertexts $C_i = E_K(M_i)$'s together, an encryption algorithm which uses cipher block chaining (CBC) computes

$$C_i = E_K(C_{i-1} \oplus M_i)$$

which ensures that each ciphertext block is unique, even when the M_i's match. The ciphertexts can now be safely concatenated into an encrypted message.

Another way of accomplishing this goal is by introducing an element of randomness to the process. The following scheme uses this idea to simulate the application of a one-time pad to each block, each time with a different "random" key.

Output feedback mode: (OFB) A random bitstring called the *initialization vector*, denoted IV, is chosen. We use the random IV to produce a set of "random" one-time pad keys:

$$Z_i = \begin{cases} IV, \text{ if } i = 0, \\ E_K(Z_{i-1}), \text{ otherwise} \end{cases}$$

We can then encrypt the message with

$$C_i = Z_i \oplus M_i$$

and concatenate the ciphertexts into the encrypted message, with the first block being the random initialization vector, so the recipient can decrypt the message.

One final example draws on the same ideas as the first two, but with the ability to parallelize the algorithm in mind, a quality which both CBC and OFB lack.

Counter mode: Similar to OFB, an initialization vector IV is chosen. Rather than chain blocks together, a simple counter is used:

$$Z_i = E_K(IV + i), C_i = Z_i \oplus M_i$$

3 Asymmetric Key Cryptography

The major flaw with symmetric key cryptography is that the two parties need to have a pre-determined secret key. This poses a chicken-and-egg problem - in order to have a secret key, the two parties must have a secure channel, which is only possible, under symmetric key cryptography, with an existing secret key. In practice, there are many situations in which two parties have never communicated before or have no shared secret key between them or no access to a secure channel, so it becomes necessary to find ways of securely transmitting information over insecure channels. *Public key cryptography* is a subset of asymmetric cryptography, consisting of a cryptographic algorithms which rely on the parties each having a *public*

key, which is not confidential and can be shared with anyone, and a *private key*, which no one else can have; the crux of public key cryptography is that the public and private keys should be cryptographic inverses - each should be able to decrypt a message encrypted with the other. The basic idea then is that if Alice wants to send a message to Bob, she can use Bob's public key to encrypt the message, and only Bob, who has the private key, can decrypt it. Any eavesdropper will only have access to the message encrypted with Bob's public key, and is unable to decrypt the message without Bob's private key, which no one but Bob has. The only design flaw with this scheme is that Alice needs some secure way of obtaining Bob's public key in the first place. In practice, this isn't too big of a concern since Bob's public key is readily accessible publicly and can be broadcast to Alice.
The most common use of asymmetric cryptography is to enable two parties to agree on a secret symmetric key over an insecure channel, so they may proceed with symmetric key cryptography from that point onwards. One way of doing this is with Diffie-Hellman key exchange.

Diffie-Hellman key exchange: The basic idea behind Diffie-Hellman is that if we have access to an associative (over a binary operator $\cdot$) and one-way (easy to compute but difficult to invert) function f, then we can use the following scheme.

1. Alice and Bob choose respective private keys K_A and K_B, unknown to anyone but themselves. They also agree on a public key K, accessible to anyone.

2. Alice computes $a := f(K_A \cdot K)$, and sends it to Bob. Bob computes $b := f(K_B \cdot K)$ and sends it to Alice.

3. Alice computes $f(K_A, b)$ while Bob computes $f(K_B, a)$. Since f is associative, we have

$$K' = f(K_A, b) = f(K_A, f(K_B \cdot K)) = f(f(K_A \cdot K_B) \cdot K) = f(K_B, a)$$

   which ensures that Alice and Bob both arrive at the same K', which can be used as a symmetric key in future communication.

The key K' is secure because the only information an eavesdropper Eve has access to is $K, f(K_A, b), f(K_B, a)$. Since f is difficult to invert, it's difficult to obtain either a or b given these three items and without knowing K_A or K_B.

There are many possible implementations of the above scheme; one common one uses the difficulty of the discrete logarithm problem (solving the equation $x^k = y$ for x, over a finite group) to produce the one-way function f. The finite group used here is the multiplicative group modulo a prime; the agreed upon public key is a pair $K = (p, g)$, where p is a large prime used as the modulus of the finite group, and g is some number in the group which we'll use as the base of the exponent. Alice's and Bob's secret keys are, respectively, random elements a, b of the finite group. Alice computes $A = g^a (modp)$, and Bob computes $B = g^b (modp)$. Then, once the two are exchanged, we have $K' = A^b = g^{ab} = g^{ba} = B^a (modp)$, but the difficulty of the discrete logarithm problem implies that it's difficult to deduce g^{ab} from g^a, g^b, g, p.

Notice that while Diffie-Hellman is asymmetric cryptography, it isn't quite public key cryptography, since it can't be used to directly encrypt messages. If we wish to directly encrypt messages for transit over an insecure channel, we can adapt Diffie-Hellman into a public key encryption scheme, called El Gamal encryption.

El Gamal encryption: If Alice wants to send a message to Bob, then in accordance with public key cryptography, she would encrypt the message with Bob's public key. Bob chooses g, p as in Diffie-Hellman, keeping them secret, and uses a random element of the finite group b as his private key. Bob's public key is $B = g^b \pmod p$. Alice can then encrypt her message m by choosing a random element $r \neq p - 1$ and send Bob the pair $(g^r \pmod p), m \cdot B^r \pmod p))$. Then Bob can find m by computing $(g^r)^{-b}$ and multiplies by $(m \cdot B^r)$ to obtain $B^{-r} \cdot m \cdot B^r = m \pmod p$ as desired. An eavesdropper Eve will only have the information $g^r, m \cdot B^r, B$, from which b or r can't be extracted.

Since r is chosen randomly each time, this is similar to a modified one-time pad.

4 Integrity

Our discussion on public key cryptography raised an important concern - for Alice to send a message to Bob over an insecure channel, she would first need to acquire Bob's public key. There are no issues with confidentially transmitting Bob's public key, as this is public information, but under the right circumstances an attacker might be able to trick Alice into thinking a false key is Bob's public key. Thus, while confidentiality isn't a concern, *integrity* is.
In contrast to confidentiality, we'll look at schemes for providing authenticity and integrity. Authenticity is a guarantee that a message is coming from who we think it is. Integrity is a guarantee that the message wasn't modified on transit, i.e. that the message we receive is the same message that was sent. *Spoofing* messages is sending messages under the guise that the messages come from someone else, violating authenticity, while *tampering* messages is modifying existing messages, violating integrity.
One way to leverage symmetric key cryptography to provide both integrity and authentication is using a *message authentication code*, or MAC. The idea is that Alice will use a symmetric key to encrypt a piece of information that Bob could verify,

ensuring that the message came from someone who had the secret symmetric key, which is only Alice, providing authenticity. The information we encrypt is a checksum, or some form of invertible function on the message, so that Bob could recompute the checksum upon receiving the message and check it against the sent checksum; if the message had been tampered with, the checksums would no longer match, providing integrity. Of course, an eavesdropper Eve could modify the message as well as the checksum to ensure that they match, but in the case of a MAC, since the checksum is encrypted with the secret symmetric key, Eve, who doesn't have the key, is unable to recompute the correct MAC needed to fool Bob. Thus, MACs are essentially checksums encrypted with a symmetric key. A precise definition follows.

Message authentication code: If Alice is to send a message M to Bob using a symmetric key K, she can guarantee integrity and authenticity by using some publicly known function f to compute a *tag*: $f(M,K)$. She then sends Bob the pair $(M, f(M,K))$; Bob, who has the key K, can re-compute $f(M,K)$ and see if it matches the received tag.

Of course, in practice, M is also encrypted to ensure confidentiality, so typically Alice will send over $(E_K(M), f(E_K(M), K))$, where E_K is the encryption algorithm. Ideally, f should be a "pseudo-random" function in the same sense that E_K is pseudo-random, so that Eve can't guess the correct tag for a modified message of her design. They should also have a zero, or very low, probability of colliding, so that we can be reasonably certain that $M \neq M' \to f(M,K) \neq f(M',K)$. We can usually use block ciphers as MACs. A MAC is secure if it can survive chosen-plaintext attacks.

If a shared symmetric key is not available, we can use public key cryptography instead to ensure authenticity and integrity. This is done with a *digital signature*. The idea is that if Alice encrypts the message with her private key, called "signing" it, which only she has, then Bob can verify this by using Alice's public key to decrypt the information and validate it; anyone trying to spoof or tamper with the message will need to re-sign the modified message, but is unable to do so without Alice's private key.

Digital signature: If Alice is to send a message M to Bob, she can generate a private key K and a public key P. Alice then sends $(M, E_K(M))$ to Bob, who can compute $D_P(M)$ and see if it matches M. When confidentiality is also required, an encrypted version of M is used instead of the plaintext.

Note that this requires Bob to somehow have Alice's public key P, and be certain that it's unmodified and indeed Alice's public key, posing another chicken-and-egg problem.

Let's outline one very common digital signature scheme - RSA. The algorithm makes use of trapdoor functions, defined below.

Trapdoor function: A function f associated with a private key K, with the property that computing f is easy, but inverting f without K is hard. This is equivalent to saying that while computing $f(\cdot)$ and computing $f_K^{-1}(\cdot)$ is easy, computing $f^{-1}(\cdot)$ is hard.

The RSA algorithm requires a public key P and private key K, which correspond to a trapdoor function f. Given a message M, the algorithm essentially signs M with a value S such that $M = f_P(S)$. The signer, who has the private key K, computes S with $f_K^{-1}(M)$, which is easy with K, but without K can't be done faster than brute force. In practice, to keep M secret, instead of using the plaintext M we use an encrypted version or cryptographically hashed version of M. The trapdoor function f used by RSA relies on a fair amount of elementary number theory, and is difficult to invert due to the difficulty of the factoring problem.
To implement RSA, we'll use a pair of numbers (n, e) as the public key, where n factors into two (large) primes, p, q. We choose a secret key d with the property that $d = e^{-1} \pmod{\phi(n)}$, where ϕ is Euler's totient function. The trapdoor function we use is $f(x) = x^e \pmod{n}$); the reason we enforce the constraint that $ed = 1 \pmod{\phi(n)}$ is so that the function $g(x) = x^d \pmod{n}$ is the inverse of f, which is what makes f a trapdoor function (due to the discrete logarithm problem, inverting $x^e = y \pmod{n}$ is difficult, but if we have the private key d, we can apply g to f and invert it easily):

$$ed = 1 \mod \phi(n) \to ed = 1 + k\phi(n) \text{ for some } k \to g(f(x)) = (x^e)^d = x^{ed} = x^{1+k\phi(n)} = x \cdot (x^{\phi(n)})^k = x \cdot 1^k = x \pmod{n}$$

where $\gcd(x,n) = 1 \to x^{\phi(n)} = 1 \pmod{n}$ by Euler's generalization of Fermat's little theorem. The reason we require that $n = pq$ is that this makes computing d easy - simply compute $\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$, after which we can compute e^{-1} using the extended Euclidean algorithm. However, given n, e, m^e, it's difficult to recover d. Note that if the attacker can factor $n = pq$ then she can compute d the same way Alice would, so the security of RSA depends on the difficulty of the factoring problem. Also note that if it's not the case that $\gcd(m,n) = 1$, we'd have to pad m using a pre-determined padding scheme to force $\gcd(m,n) = 1$.

5 Key Management and Password Hashing

In contrast to the previous notes, in this section we'll consider the secure use of passwords and how to manage symmetric and asymmetric keys safely. Due to the ubiquity of passwords and the fact that they are chosen by laypeople rather than

by secure algorithms, passwords are a critical point of study in security. Additionally, because most people use the same passwords for many accounts, breaking a single password is often equivalent to breaking several.
Weaknesses of passwords:

1. *Brute force guessing*: If a password is short in length or follows easy-to-guess or statistically common conventions, an attacker might be able to guess it quickly simply by trying combinations.

2. *Social engineering*: Tricking a user into divulging the password or information that can be used to recover the password (e.g. answers to secret questions).

3. *Eavesdropping*: If passwords are transmitted over an insecure channel, an attacker might be able to intercept them.

4. *Client-side malware*: Malicious software installed on a user's computer (e.g. keyloggers) can often reveal passwords.

5. *Server compromise*: If the data on a server storing passwords is leaked, any attacker would gain knowledge of every user's password.

Social engineering involves different kinds of attacks than the ones we've been studying and is prevented with strong design philosophy, whereas client-side malware is often simply impossible to stop until detected (two-factor authentication is one defense, but is often inconvenient) - the focus is usually on preventing it from being installed in the first place. We'll present ways of mitigating the other three vulnerabilities.

Eavesdropping: Protecting against eavesdropping is the simplest; we can simply encrypt passwords and any other data between a client and a server. Nonetheless, there are edge cases that need to be considered.

Brute force: Although the search space for ASCII string of length n grows exponentially, the majority of people do not use random passwords, and tend to, due to security ignorance and ease of use, use many of the same passwords or password conventions. The following statistics illustrate this:

About 1% of users use the top 10 most common passwords, and about 50% use the top million.

Statistically, the possibilities can be narrowed down even further if the attacker can gain significant personal information about a particular user, such as birthdays, name of spouse, etc. With modern computing power, guessing a million passwords is far from infeasible. It's necessary here to draw a distinction between the kinds of attacks we're considering:

- **Targeted attack**: Attacks in which a specific user's password is desired.
- **Untargeted attack**: Attacks in which the attacker simply wants some users' passwords (usually as many as possible).This attack is much easier than the first due to its less stringent requirements.

Targeted attacks can be effectively mitigated by combating computing power by imposing rate limits which force the attacker to guess at a certain speed, allowing us to introduce a typically imperceptible or negligible delay the user enters his password, but which multiplies the total amount of time to guess the password by a large factor, making it intractable (commonly, websites will enforce "lock-outs" for a pre-set period of time, which commence after a certain number of incorrect login attempts, or simply force each login attempt to take a non-negligible amount of time; nonetheless, many websites still don't do this). These techniques don't protect against untargeted attacks, however, as the attacker can still leverage the above statistics if he can access many accounts (e.g. guessing the top 10 most common passwords once per a thousand accounts won't trigger any rate limits but will expect to break one account).

Server compromise: Similar to the brute-force case in that a server's data is leaked, exposing its users' passwords. If the passwords are stored in plaintext on the server, nothing can be done and all the passwords have been leaked. It's advisable to only store the cryptographic hashes of the passwords, and only verify the password hash upon login, rather than the password itself. However, the statistics presented above and the fact that rate limits and other defenses of the like since the attacker can run any brute-force methods on the encrypted passwords on his own machine pose complications. If the hash used can be computed quickly, such as SHA-256, often attackers can make guesses as fast as one billion times per second, making it very feasible to guess the top one million password hashes for each user. Since many of the top million passwords can be expected to overlap, clever algorithms can further cut the time down by maintaining a sorted list of the hashes of the million passwords and simply binary searching it for each user. These *amortized time attacks* can be prevented by including a random salt that's appended to and encrypted with the password, ensuring that no two password hashes overlap. The best way to mitigate the first attack is to use a slow hash function, which can linearly scale how long it takes an attacker to break a password, at the expense of an often imperceptible delay to users during login.