Modern Cryptography, Part IV

Diffie-Hellman key exchange

All of the ciphers we've covered up to this point need both sides of the communication to have a key. In the old days, people could meet up and exchange keys in person, but that won't work for the internet. The solution, discovered in the 1970s, surprised many people at the time. It's called *Diffie-Hellman key exchange*. It involves two parties communicating in public, yet being able to generate a shared secret number that only they will know.

Before we describe the way it works, here is a simpler, but insecure, idea. We will assume the two people communicating are Alice and Bob. Alice generates a random number a and sends it to Bob. Bob generates his own random number b and sends it to Alice. They then both multiply the numbers to get ab. This number would be the shared secret. The problem, of course, is that it's not secret since someone watching the communication could see a and b and thus compute ab themselves. But at least the value of ab is a value that both Alice and Bob now have, and they could use that as a key if they were sure no one else observed their communication.

It would be good to find a way to disguise a and b before sending them. The trick that Diffie-Hellman uses is to disguise a by doing $g^a \bmod p$, where p is a large prime number and g is some other number. Raising a number to the a power and then modding by a prime seems to be a hard thing to reverse. As long as the prime is very large, no one has yet found a good way to do that.

Here is an example of the key exchange using $g = 3$ and $p = 7927$. First, Alice and Bob pick random numbers. Suppose Alice picks $a = 45$ and Bob picks $b = 144$. Alice disguises her number by computing $g^a \bmod p$, which is $A = 3^{45} \bmod 7927 = 3147$. Bob disguises his number by computing $B = g^b \bmod p$, which is $3^{144} \bmod 7927 = 1975$.

Alice then computes the shared secret by doing $B^a \bmod p$, which is $1975^{45} \bmod 7927 = 3641$. Bob computes the shared secret by doing $A^b \bmod p$, which is $3147^{144} \bmod 7927 = 3641$. Notice that they both get the same value, 3641. This is the shared secret.

Some notes about the process:

- Notice that $A^b = (g^a)^b = g^{ab}$ and $B^a = (g^b)^a = g^{ab}$, which are the same. It turns out that the mods won't affect this. So this is why Alice and Bob both get the same value at the end.
- If you're trying this out yourself, Python's `pow` function can be used. `pow(x,y,z)` does $x^y \bmod z$.
- An eavesdropper can do a brute-force guess-and-check to try to figure out what a and b are. If they can do that, then they can figure out the shared secret. To prevent this, p must be chosen to be incredibly large. People recommend that p be at least 2048 bits, which corresponds to a number over 600 digits long.
- The values of p and g need to be carefully chosen. There are a few good values that people often repeatedly use.
- Diffie-Hellman is subject to a man-in-the-middle attack. This is where Eve, the eavesdropper, pretends to be Bob to Alice and pretends to be Alice to Bob. Eve ends up creating shared secrets with Alice and Bob separately. Each of them thinks they are communicating with the other, but they are really communicating with Eve. To avoid this, Diffie-Hellman needs to be combined with an authentication scheme so that Alice and Bob can be sure they are communicating with each other and not an impersonator.
- Once the shared secret is generated, Alice and Bob use it to create keys to use with an ordinary cipher, such as AES.
- Plain Diffie-Hellman is falling out of use in favor of a *elliptic curve Diffie-Hellman* The overall idea is the same, but instead of using modular arithmetic, it uses a different type of arithmetic based on elliptic curves. The math behind elliptic curves is typically covered in an upper level undergraduate abstract algebra course.

Public-key cryptography

All of the ciphers we have seen so far are called *symmetric*. They are called this because both sides of the communication use the same key. There is another type of cryptography called *asymmetric*, in which different keys are used for encryption and decryption. Another name for this is *public-key cryptography*. One of the keys, the *public key*, is available to the public, and the other, the *private key*, is kept secret.

Here is the basic idea: Suppose Alice wants people to be able to send encrypted messages to her using public-key cryptography. She creates a public-private key pair. She publishes the public key and keeps the private key secret. If Bob wants to send an encrypted message to Alice, he encrypts it using the public key. Alice uses her private key to decrypt it. The public and private keys are tied together so that this works.

A nice analogy for this would be if Alice creates physical padlocks and keys. If someone wants to safely send something to Alice, they get a padlock from Alice and put their item in a box locked with Alice's padlock. Alice can use her key to open the lock. The padlock is like the public key and Alice's physical key is like the private key.

Trapdoor functions

The security of the Diffie-Hellman key exchange relies on the fact that it is easy to compute $g^a \bmod p$, but it is hard to reverse the process to figure out a. This mathematical operation is an example of a *trapdoor function*, something that is easy to do in one direction and hard to do in the other. Trapdoor functions are very important in modern cryptography.

A simple, but powerful, trapdoor function is multiplication of prime numbers. It's easy to multiply to prime numbers, but it is hard to "unmultiply" them, i.e., to factor them. For instance, it is easy to multiply 999979×999983, but it is much harder to factor their product 999962000357. This trapdoor is the basis of one of the most well-known types of public-key cryptography—RSA.

RSA

RSA is named for the three people who created it: Rivest, Shamir, and Adleman. It's probably easiest to describe with an example. We'll use very small prime numbers in this example to demonstrate the concept, but in the real world, very large primes need to be used.

Alice starts by generating a public-private key pair. She does this by choosing two prime numbers, p and q. Let's use $p = 101$ and $q = 137$. She then multiplies them together to get $n = pq = 13837$. She next chooses a value e that has no factors in common with $p-1$ or $q-1$. In this case, $e = 23$ works. Finally, she computes a value d that has the property that $de \bmod (p-1)(q-1) = 1$. There is a very fast algorithm, the Extended Euclidean Algorithm, that can be used to do this. We won't go into the details here. The value of d turns out to be 7687 in this example. The public key that Alice publishes is n and e. The private key is d. Alice also shouldn't let anyone know what p and q are.

Let's say Bob wants to encrypt a message with Alice's private key. RSA only works with numbers, so Bob would have to convert his message to a sequence of numbers. There are many ways to do this, such as ASCII or Unicode character codes. Let's say Bob's message is the number $M = 65$. he computes $C = M^e \bmod n$, which is $65^{23} \bmod 13837 = 12429$. To decrypt, Alice does $C^d \bmod n$, which is $12429^{7687} \bmod 13837$. This comes out to 65, as it should. If you want to try this out, use Python's `pow` function (`pow(x,y,z)` does $x^y \bmod z$). Below are a few notes about RSA.

- Just like with Diffie-Hellman, the primes must be very large, at least 2048-bits (over 600 digits long) to prevent brute-force and similar attacks.
- RSA is notoriously difficult to implement correctly. Even though the process is pretty simple, there are lots of subtle details. For instance, the plaintext must be disguised before encrypting for similar reasons to why ECB mode is bad. Another thing is that the same n must not be reused with multiple values of e. There are

a dozen other such gotchas. Partly because of this, RSA is slowly being phased out in favor of elliptic curve methods.

- RSA is very slow. It's not typically used to encrypt large amounts of information. Its primary use is for digital signatures, which we will cover later.
- RSA's security is based on the fact that it seems to be hard to factor large numbers. If someone could figure a way to decompose n into its component primes p and q, they could figure out the secret key d. Experts think that factoring is a hard problem, but no one is really sure. One thing that is sure: if people ever figure out how to build a quantum computer, then RSA is toast, as quantum computers can factor large numbers quickly.

Side-channel attacks

The encryption algorithms we have covered (DES, AES, RSA) are all mathematically pretty sound. The weakness is the programmers who implement the algorithms in code. If a programmer is not careful, they can introduce ways for an attacker to figure out a secret key without breaking the algorithm itself. Attacks of this sort, that rely on information obtained from observing the algorithm running, are called *side-channel attacks*. Some of them are especially devious.

To give a sense for how these work, consider a simple algorithm that tests if an input string matches a secret string. Here is pseudocode for what that string comparison actually does.

```
if length(input) != length(secret0:
    return false
else:
    Loop over the characters of the input:
        if current character of input != current character of secret:
            return false
return true
```

If the two strings have a different length, then it will immediately return false. If they are the same length, then it goes through comparing character-by-character. So for strings of different length, the string comparison will be quicker than if they are of the same length. An attacker could try inputs of varying length until they find one for which the comparison takes noticeably longer. That will tell them the length. Then they start guessing starting characters A, B, C, or whatever until they get a comparison that takes longer. That indicates that they have the first character right (because then the program has to take time to check the second character, whereas it returns false right away if the first characters don't match). They continue this way, guessing the second, third, etc. letters, until they have the whole string. The side channel here is the time it takes the string comparison algorithm to perform its job. To fix the problem, some code has to be added to make sure that all comparisons take the same amount of time.

This type of timing side channel can be used to break strong algorithms, like AES and RSA. For instance, in RSA, when we compute $C^d \bmod n$, the algorithm that does the raising to a power does its work based on the binary structure of the secret key d, doing more work for 1s and less work for 0s. Timing analysis can be used to figure out the key from there.

Also, since the CPU is working harder on those 1s than on the 0s, its power output varies. People can attach some sensors to measure the power output of the CPU and use that to figure out the key. An especially wild variation of this is acoustic cryptanalysis, which uses the high-pitched sounds the CPU puts out to figure out when it is working harder than other times. These attacks are not theoretical; they have actually been implemented.

Another interesting attack is AES cache timing. Here the term cache refers to an area of fast memory on the CPU. Memory accesses to things in the cache take less time than accesses to things stored in RAM. If one process does an AES encryption, after it's done, the cache will contain some values related to the encryption. Programs can't read the cache directly, but they can use the fact that cache accesses take less time than regular ones. This is called, probing, where the program will try out certain values and note how long it takes to access them. With enough of these probes, it can make a reasonable guess as to some things that are in the cache and figure out the key from there. One place this can happen is on a virtual private server (VPS). Often a single cloud computer hosts multiple different websites. A process running on behalf of one of those sites can use cache probing to figure out AES keys based on information left in the cache by other sites.

This is just a small sampling of side-channel attacks. There are many others based on things such as keystroke timing and electromagnetic radiation leaks from monitors.