SAFE PRIMES IN DH KEY EXCHANGE

Diffie-Hellman key agreement protocol uses modular exponentiation and calls for use of special prime numbers. If you ever wondered why, I'll try to explain.

Diffie-Hellman key exchange

The "classical" Diffie-Hellman key exchange also known as Finite Field Diffie-Hellman uses one type of operation — modular exponentiation — and two secrets for two communication peers to arrive at a single shared secret.

The protocol requires a prime number and a number that is a so-called "generator" number to be known to both peers. This is usually achieved by either the server sending those values to the client (e.g. in TLS before v1.3 or in some SSH key exchange types) or by distributing them ahead of time to the peers (e.g. in IPSec and TLS 1.3).

When both peers know which parameters to use, they generate a random number, perform modular exponentiation with this random number, group generator and prime and send the result to the other party as a "key share". That other party takes this key share and uses it as the generator to perform modular exponentiation again. The result of that second operation is the agreed key share.

If we define g as the generator, p as the prime, S_r as the server selected random, S_k as the server key share, C_r as the client selected random and C_k as the client key share, and SK being the agreed upon secret, the Server performs following operations:

$g^{S_r} = S_k \bmod p$

$C_k^{S_r} = SK \bmod p$

while the Client performs following operation:

$g^{C_r} = C_k \bmod p$

$S_k^{C_r} = SK \bmod p$

because

$(g^{C_r})^{S_r} = (g^{S_r})^{C_r} = SK$ both parties agree on the same SK.

Unfortunately both peers need to operate on a value provided by the other party (not necessarily trusted or authenticated) and their secret value at the same time. This calls for the the prime number used to have some special properties.

Modular exponentiation

The basic operation we'll be dealing with is modular exponentiation. The simple way to explain it is that we take a number, raise it to some power. Then we take that result and divide it by a third number. The remainder of that division is our result.

For `2^10` mod 12, the calculation will go as follows, first exponentiation:

2^10 = 1024

Then division:

1024 = 85*12 + 4

So the result is 4.

One of the interesting properties of modular exponentiation, is that it is cyclic. If we take a base number and start raising it to higher and higher powers, we will be getting the same numbers in the same order:

```
$ python powmod.py -g 3 -m 14 -e 28
3^0  mod 14 = 1
3^1  mod 14 = 3
3^2  mod 14 = 9
3^3  mod 14 = 13
3^4  mod 14 = 11
3^5  mod 14 = 5
3^6  mod 14 = 1
3^7  mod 14 = 3
3^8  mod 14 = 9
3^9  mod 14 = 13
3^10 mod 14 = 11
3^11 mod 14 = 5
3^12 mod 14 = 1
3^13 mod 14 = 3
3^14 mod 14 = 9
3^15 mod 14 = 13
3^16 mod 14 = 11
3^17 mod 14 = 5
3^18 mod 14 = 1
3^19 mod 14 = 3
3^20 mod 14 = 9
3^21 mod 14 = 13
3^22 mod 14 = 11
3^23 mod 14 = 5
3^24 mod 14 = 1
3^25 mod 14 = 3
3^26 mod 14 = 9
3^27 mod 14 = 13
```

This comes from the fact that in modulo arithmetic, for addition, subtraction, multiplication and exponentiation, the order in which the modulo operations are made does not matter; `a + b mod c` is equal to `(a mod c + b mod c) mod c`. Thus if we try to calculate the example for `3^17 mod 14` we can write it down as `((3^6 mod 14) * (3^6 mod 14) * (3^5 mod 14)) mod 14`. Then the calculation is reduced to `1 * 1 * 3^5 mod 14`.

The inverse of modular exponentiation is discrete logarithm, in which for a given base and modulus, we look for exponent that will result in given number:

g^x mod m = n

Where g, m and n are given, we're looking for x.

Because there are no fast algorithms for calculating discrete logarithm, is one of the reasons we can use modulo exponentiation as the base of Diffie-Hellman algorithm.

Cyclic groups

Let's see what happens if we start calculating results of modular exponentiation for 14 with different bases:

```
$ python groups.py -m 14
Groups modulo 14
g:  0, [1, 0]
g:  1, [1]
g:  2, [1, 2, 4, 8]
g:  3, [1, 3, 9, 13, 11, 5]
g:  4, [1, 4, 2, 8]
g:  5, [1, 5, 11, 13, 9, 3]
g:  6, [1, 6, 8]
```

```
g:  7, [1, 7]
g:  8, [1, 8]
g:  9, [1, 9, 11]
g: 10, [1, 10, 2, 6, 4, 12, 8]
g: 11, [1, 11, 9]
g: 12, [1, 12, 4, 6, 2, 10, 8]
g: 13, [1, 13]
```

Neither of the numbers can generate all of the numbers that are smaller than the integer we calculate the modulo operation with. In other words, there is no generator (in number theoretic sense) that generates the whole group.

To find such numbers, we need to start looking at prime numbers.

Cyclic groups modulo prime

Let's see what happens for 13:

```
$ python groups.py -m 13
Groups modulo 13
g:  0, [1, 0]
g:  1, [1]
g:  2, [1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7]
g:  3, [1, 3, 9]
g:  4, [1, 4, 3, 12, 9, 10]
g:  5, [1, 5, 12, 8]
g:  6, [1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11]
g:  7, [1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2]
g:  8, [1, 8, 12, 5]
g:  9, [1, 9, 3]
g: 10, [1, 10, 9, 12, 3, 4]
g: 11, [1, 11, 4, 5, 3, 7, 12, 2, 9, 8, 10, 6]
g: 12, [1, 12]
```

The obvious result is that we now have 4 generators — 2, 6, 7 and 11 generate the whole group.

But there is other result hiding. Let's see the results for 19, but with sizes of those groups shown:

```
$ python groups-ann.py -m 19
Groups modulo 19
g:   0, len:   2, [1, 0]
g:   1, len:   1, [1]
g:   2, len:  18, [1, 2, 4, 8, 16, 13, 7, 14, 9, 18, 17, 15, 11, 3, 6, 12, 5, 10]
g:   3, len:  18, [1, 3, 9, 8, 5, 15, 7, 2, 6, 18, 16, 10, 11, 14, 4, 12, 17, 13]
g:   4, len:   9, [1, 4, 16, 7, 9, 17, 11, 6, 5]
g:   5, len:   9, [1, 5, 6, 11, 17, 9, 7, 16, 4]
g:   6, len:   9, [1, 6, 17, 7, 4, 5, 11, 9, 16]
g:   7, len:   3, [1, 7, 11]
g:   8, len:   6, [1, 8, 7, 18, 11, 12]
g:   9, len:   9, [1, 9, 5, 7, 6, 16, 11, 4, 17]
g:  10, len:  18, [1, 10, 5, 12, 6, 3, 11, 15, 17, 18, 9, 14, 7, 13, 16, 8, 4, 2]
g:  11, len:   3, [1, 11, 7]
g:  12, len:   6, [1, 12, 11, 18, 7, 8]
g:  13, len:  18, [1, 13, 17, 12, 4, 14, 11, 10, 16, 18, 6, 2, 7, 15, 5, 8, 9, 3]
g:  14, len:  18, [1, 14, 6, 8, 17, 10, 7, 3, 4, 18, 5, 13, 11, 2, 9, 12, 16, 15]
g:  15, len:  18, [1, 15, 16, 12, 9, 2, 11, 13, 5, 18, 4, 3, 7, 10, 17, 8, 6, 14]
g:  16, len:   9, [1, 16, 9, 11, 5, 4, 7, 17, 6]
g:  17, len:   9, [1, 17, 4, 11, 16, 6, 7, 5, 9]
g:  18, len:   2, [1, 18]
```

Note that all the sizes of those groups are factors of 18 – that is p-1.

The third observation we can draw from those results is that, for any number the size of the group of the generated elements will be at most as large as the size of the base number.

With 19, if we take generator 8, the size of its subgroup is 6. But size of subgroups of 7, 18, 11 and 12 is respectively 3, 2, 3 and 6.

Thus, not only is the subgroup much smaller than the full group, it is also impossible to “escape” from it.

Safe primes

We saw that for primes, some bases are better than others (look into finite fields to learn more).

As we noticed, sizes of all groups are factors of the prime less one (see Fermat’s little theorem for proof of this). Of course, with the exception of 2, all primes are odd numbers, so p-1 will always be divisible by two – it will be a composite number. But q = (p-1)/2 doesn’t have to be composite. Indeed, we call primes for which q is also prime safe primes.

Let’s see what happens if we calculate groups of such a prime:

```
$ python groups-ann.py -m 23
Group sizes modulo 23
g:   0, len:   2, [1, 0]
g:   1, len:   1, [1]
g:   2, len:  11, [1, 2, 4, 8, 16, 9, 18, 13, 3, 6, 12]
g:   3, len:  11, [1, 3, 9, 4, 12, 13, 16, 2, 6, 18, 8]
g:   4, len:  11, [1, 4, 16, 18, 3, 12, 2, 8, 9, 13, 6]
g:   5, len:  22, [1, 5, 2, 10, 4, 20, 8, 17, 16, 11, 9, 22, 18, 21, 13, 19, 3, 15, 6, 7, 12, 14]
g:   6, len:  11, [1, 6, 13, 9, 8, 2, 12, 3, 18, 16, 4]
g:   7, len:  22, [1, 7, 3, 21, 9, 17, 4, 5, 12, 15, 13, 22, 16, 20, 2, 14, 6, 19, 18, 11, 8, 10]
g:   8, len:  11, [1, 8, 18, 6, 2, 16, 13, 12, 4, 9, 3]
g:   9, len:  11, [1, 9, 12, 16, 6, 8, 3, 4, 13, 2, 18]
g:  10, len:  22, [1, 10, 8, 11, 18, 19, 6, 14, 2, 20, 16, 22, 13, 15, 12, 5, 4, 17, 9, 21, 3, 7]
g:  11, len:  22, [1, 11, 6, 20, 13, 5, 9, 7, 8, 19, 2, 22, 12, 17, 3, 10, 18, 14, 16, 15, 4, 21]
g:  12, len:  11, [1, 12, 6, 3, 13, 18, 9, 16, 8, 4, 2]
g:  13, len:  11, [1, 13, 8, 12, 18, 4, 6, 9, 2, 3, 16]
g:  14, len:  22, [1, 14, 12, 7, 6, 15, 3, 19, 13, 21, 18, 22, 9, 11, 16, 17, 8, 20, 4, 10, 2, 5]
g:  15, len:  22, [1, 15, 18, 17, 2, 7, 13, 11, 4, 14, 3, 22, 8, 5, 6, 21, 16, 10, 12, 19, 9, 20]
g:  16, len:  11, [1, 16, 3, 2, 9, 6, 4, 18, 12, 8, 13]
g:  17, len:  22, [1, 17, 13, 14, 8, 21, 12, 20, 18, 7, 4, 22, 6, 10, 9, 15, 2, 11, 3, 5, 16, 19]
g:  18, len:  11, [1, 18, 2, 13, 4, 3, 8, 6, 16, 12, 9]
g:  19, len:  22, [1, 19, 16, 5, 3, 11, 2, 15, 9, 10, 6, 22, 4, 7, 18, 20, 12, 21, 8, 14, 13, 17]
g:  20, len:  22, [1, 20, 9, 19, 12, 10, 16, 21, 6, 5, 8, 22, 3, 14, 4, 11, 13, 7, 2, 17, 18, 15]
g:  21, len:  22, [1, 21, 4, 15, 16, 14, 18, 10, 3, 17, 12, 22, 2, 19, 8, 7, 9, 5, 13, 20, 6, 11]
g:  22, len:   2, [1, 22]
```

The groups look very different to the ones we saw previously, with the exception of bases 0, 1 and p-1, all groups are relatively large – 11 or 22 elements.

One interesting observation we can make about bases that have group order of 2q, is that even exponents will remain in a group of size 2q while odd will move to a group with order of q. Thus we can say that use of generator that is part of group order of 2q will leak the least significant bit of the exponent.

That’s why protecting against small subgroup attacks with safe primes is so easy, it requires comparing the peer’s key share against just 3 numbers. It’s also the reason why it’s impossible to “backdoor” the parameters (prime and generator), as every generator is a good generator.