Diffie-Hellman Revisited

Introduction

Consider the situation where a customer is using an internet banking application to connect to a bank.

If a hacker interposes a proxy between the application and the bank, he can wait for the customer to present his login credentials, be approved by the authentication system – however complex and secure that might be - then drop the customer's connection.

From this point, the hacker can perform any financial transactions as if he were the customer.
This is not a good scenario, so it is necessary to provide an encrypted communication channel, which is proof against a man-in-the-middle who can capture the encryption keys, passed between the customer and the bank.

The Diffie-Hellman public key exchange protocol is perfectly suited to such an application, since it is totally proof against any observation of the communications traffic and, if the attacker attempts to take the place of the customer after the successful login, he will be unable to correctly encrypt or decrypt any subsequent messages.

Unfortunately, the algorithm employed by standard implementations of Diffie-Hellman is extremely compute-intensive, and not easily deployed on a commonly available server. The numbers involved require thousands of bits, and the computation of some really ugly prime numbers. Most everyday computers will be incapable of calculating numbers like 23 to the power 729, and the setup procedure can take as long as 20 minutes, which is probably longer than anyone would want to spend paying a few bills.

My objective was to simplify the algorithm, without compromising its security, in such a way that it could run on a standard 64-bit Sun server, and establish a secure, encrypted channel in a few milliseconds, without overflowing the machine's registers. This would be advantageous if it became necessary to generate a new private key with each connection.

The Algorithm

In its classical implementation, the two parties to the communication are usually called 'Bob' and 'Alice', with a jealous spouse, 'Eve', constantly eavesdropping on the conversation.

Bob and Alice share two constants which are, preferably, prime numbers, and are occasionally referred to as 'G' and 'n'. These constants are the public keys, and are visible to Eve.

Bob generates a random number, 'x' and Alice generates a random number 'y', which are used, in conjunction with 'G' and 'n', to compute two parameters, 'A' and 'B', where Bob's parameter is A = G**x mod n, and Alice's equivalent is B = G**y mod n..

Bob then sends 'A' to Alice, and Alice sends 'B' to Bob, with Eve listening intently, and recording the values of 'A' and 'B'
As soon as Bob has received 'B', he immediately computes a private key, K1, which he calculates as K1 = B**x mod n, while Alice computes her private key, K2 = A**y mod n.

By the magic of mathematics, K1 is the same as K2, and can now be used to encrypt all subsequent communications between Bob and Alice, with the hapless Eve unable to deduce the key and thus, unable to decrypt any of it.

The Simplification

An examination of the Diffie-Hellman algorithm indicated that, by the application of some third-form schoolboy mathematics, we could get around the insurmountable problem of calculating impossibly high powers of large numbers.

So, as any third-former will tell you,

$$\log (G^{**}x) = x.\log G$$

which is the key to the whole algorithm, and is easily implemented in double-precision arithmetic, even when the numbers are quite large

This leads to the following scenario:
Bob calculates logA = x.logG and Alice calculates logB = y.logG
Bob sends logA to Alice, who now calculates her private key, logK2 = y.logA.
Alice sends logB to Bob, who only needs to calculate x.logB to get his private key, K1.
As before, the two keys are identical and we can save a mathematical operation by not taking anti-logs. Even though we are using logA and logB, instead of A and B, AES 256 just sees an encryption key, and doesn't care where it came from.

The Results

Here are some results of running the simplified algorithm with G = 23, and random prime numbers for x and y:

x = 4049: logA = 5513.63600804
y = 6029: logB = 8209.85712335
logK1: 33241711
logK2: 33241711
Hash = 3e3eb5b2561f6aba5c42b9aabe7875a771032acd02b51e86dc2d281a3d1523d8

x = 7789: logA = 10606.49811474
y = 1229: logB = 1673.56351047
logK1: 13035386
logK2: 13035386
Hash = 50baf31fff8bec57f1e23384e4cf2934371e84bf49a1b83f54f321a45427c9e0

x = 823: logA = 1120.70200904
y = 2647: logB = 3604.49358194
logK1: 2966498
logK2: 2966498
Hash = c4d5bf52c769b2f2cf394e5c8bf50e325d9cc3723120b43f4ef3202cc165d816

Conclusion

So, what happened to ‘mod n’?

The reason for the virtual uncrackability of the Diffie-Hellman protocol, is that the reverse-engineering of (G**x)**y, known as the discrete logarithm problem, is almost impossible, without huge computing resources and a considerable amount of time.
The application of the modulus operation is just another layer of computation, designed to further obfuscate the numbers.
This is probably necessary if the keys are not to change for a decade or two – which some people estimate is the time it would take a hacker to brute-force them. However, I had no such constraint.
It was more convenient to use the SHA256 hash of both K1 and K2, (which is a fast irreversible operation), directly as the key for AES256 encryption of the communication channel.