DAT 510: Assignment 2

Tomasz Gliniecki

10.10.2016

Abstract

The projects focus is on public key exchange. For this purpose Diffie-Hellman key exchange method was used together with Blum Blum Shub CSPRNG to further encrypt the generated secret key. This method lets the users exchange information about how to generate the same secret key over an unsecure channel, that can be then used to encrypt messages or files with symmetric ciphers. In this case, for demonstartation purposes AES cipher was used, and plain text is being sent back and forth between the clients in a chat-like enviroment.

Introduction

The most important topics of this projects are, Diffie-Hellman key echange, Cryptographically Secure PRNG (CSPRNG) and Symmetric ciphers. To implement this architecture, especially DH's key exchange, it is required to be able to do simple aritmetic on very big numbers in the range ¿ 1024 bit to provide sufficient security. For this purpose an open source and popular library for C/C++ called *GMP*[1] for arbitrary precision arithmetic was used. This library provides functions for aritmethic operations such as addition, multiplication exponentiation and modulo operations. The library also provides more sophisticated functions, but for the purpose of this projects and the requirements, only the simple arithmetic functions were used in implementation. Another library called *crypto++*[2] was used to make use of it's AES implementation, this library also privides a wide range of functions, but for this project only pure encryption and decryption functionality was used together with custom generated key from custom Diffi-Hellman's. Having all those tools available, it was possible to create aprototype of secure communication over an unsecured channel. The DH's key exchange was used to create pair of Public and private keys, that were then used to generate a secret seed, that was then used as an input for Blum Blum Shub (BBS) CSPRNG. The output of BBS was further used on consecutive message exchange between the clients.

Design and Implementation

The basic flow of the architecure starts with generating a big prime number. For this purpose Fermats primarity test is used. There are many tests that could be implemented, the most accurate one is a deterministic primality test. Deterministic tests are very slow when one is looking for a big prime because we have to check a every number up to $p/2$ to check if any of those is a factor of p, thi is in worst case scenario. Because of that Fermats test was more usefoul, and depending on implementation the statistical probability that the output number from Fermat's test is prime is very big. Fermat's test works like that:

$$a^{p-q} \equiv 1 \ mod \ p$$

if this statement holds for a then p is *probably* a prime. This test is done multiple times usually 20-50 times, and if it holds for different values of a we incresed the probability that p is prime. For mathematical proof see [3].
For the DH's exchange to work correctly and *securely* we need a cyclig group $\mathbb{Z}_p^*$ with p prime and it's primitive element. For this purpose, instead of finding the generator for the prime that was generated previously, we can use that prime withe the formula $p_n = 2 * p + 1$. test for the primality of the new p_n that we got, and then if that number is prime we know that the generator of this cyclic group $\mathbb{Z}_{p_n}^*$ is definetively 2. The drawback of that is that we have to generate a lot of primes untill we can find one that will give us another prime formed from that equation. This is the most time consuming part of the algorithm and by trying out different methods it was wery hard to optimize it significantly. This is also one of the crutial parts of the whole system, if we do not find the correct generator and prime pairs we will create security hole in out implementation, because even though we have a cyclic group the cardinality of it might be much smaller than we thought, thus reducing the keyspace significantly.
After the cyclic group and it's generator are found, they are publically available, now Alice and Bob from our example can use secure properties of these numbers to first choose a random number with one constraint - this has to be smaller than the public prime number, this will be their private keys, and using the formula

$$K_B = a^{K_b} mod \ p$$
$$K_A = a^{K_a} mod \ p$$

They can generate their respective public keys, that they will share with eachother over an unsecure channel. After both Alice and Bob, have exchanged the keys, they can calculate the same secret key, and only Bob and Alice can do that because the have theirs private keys. this is possible because:

$$K_{secret} = K_A^{(K_b)} mod \ p$$
$$K_{secret} = K_B^{(K_a)} mod \ p$$

Which is the same as:

$$K_{secret} = a^{K_a*(K_b)} mod \ p$$
$$K_{secret} = a^{K_b*(K_a)} mod \ p$$

So they perform the same calculation, and at the same time they are the only

two parties that know the full set of arguments to be able to perform it. It is clear from this equation that to crack the K_{secret} it is required to solve the discrete logarithm problem to find the private keys, this is virtually impossible with todays technology to perform this calculation on *big* primes, see prime number factorization on wikipedia [4]
The key that they have created can be encrypted again with Cryptographicaly Secure Pseudo random number generator. One of the most known is Blum Blum Shub CSPRNG. The algorithm is pretty simple, we need to find two big primes, that have the followin properties. Those two primes have to be congruent to 3 modulo 4 $q, p \equiv\ 3mod\ 4$. As well as they cannot divide our seed for CSPRNG, that is, in our case seed is the secret key, and the two primes cannot be factors of the key. When this is satisfied we can generate the next/new key with this formula:

$$K = K_{ab}^2 mod\ (p * q)$$

We can put then previous key throught the same function on consecutive information exchange during the session. Blum Blum Shub CSPRNG has an interesting characteristics, it is possible to calculate any value of the next "key" in the sequence see [5].
Now that Alice and Bob are quite satisfied with their keys, they choose a very big prime number, and calculated private keys, encryption keys, they can go to the last steps which are encryption and decryption.
For this purpose a symmetric cipher is used, since the clients can generate the same key securely, we can use a symmetric cipher. In this implementation AES cipher with CBF (cipher block chaining) was used. Crypto++ [2] library provided all the necessary functions to encrypt messages using the key generated with DH's key exchange. The messages are encrypted with 256bit key, that is extracted as the least significant bits from the CSPRNG key output. Blum Blum Shub has no known vulerabilities when it comes to choosing what parts of it's output will be used for encryption. This projects implementation provides an interface to choose the size of the initial prime number for the DH's, but AES will always use 256 bit as a key in this case.

Test Results

There was not much relevant testing done in this project, as the scope was kind of limited. But a lot of work has been put into trying to figure out a way for improved prime generation especially the part following the cycclic group generator for cyclic group $\mathbb{Z}_p^*$, and the results were inconclusive. One of the possible improvements could be to instead of checking only if $p_n = 2 * p + 1$

(1) is prime, use other prime numbers instead of 2 int that calculation, this could increase the chance of getting a prime by doing a loop thgroung primes $2 < r < 100$ where r is prime, and using the formula $p_n = r * p + 1$. If the chances of getting a prime p_n here are greater it is more efficient to try this instead of directly going into generating new prime and putting it into (1). There was no significan improvement with that solution so no changes were made in this particular case, generation of 2048 bit strong primes takes in worst tested case $< 200sek$, on average $5 - 10sek$.

Discussion

The generation of big strong secure primes is happening quite fast, and for the purpose of this project there is no real cons of generating 2048 bit primes on for every new communication session. Although one should have in mint that finding strong primes is the most demanding task in the whole protocol, and should be taken in to concideration on future improvements of the program. In real worls one would usually generate a very strong (big) prime number, and since it's public we really don't need to generate many of those. There is enough space in the key for everyone to produce unique private and public keypairs. But there can be some collisions, these are more probable to happen with smaller prime numbers, that adding to the fact that small primes ¡ 1024 bit might be easily cracked in the near future, see integer facorization [4]. In this implementation one can easily test and see how quickly it's possible to generate even 2048 stong primes with their generators. many companies like google and facebook use these kinds of public numbers for keyechgange with clients.
Here is presented, a sample output from the program, for purposes of the paper 256bit prime was generated, otherwice it would not fit on the side. These are also displayed in base 62, to further reduce the length.

```


Cyclic group 256bit:
fK4Iz02xqqLuRcSwuR3iRMXnbzPxEKAoF5eIVht1QxP
The generator:
2

The shared seed key K_ab:
        Alice:
Mv8qZmGWPgKojpMfNOQQQxRoF95cdO7633QsB2vQqHS
```

```
        Bobby:
Mv8qZmGWPgKojpMfNOQQQxRoF95cdO7633QsB2vQqHS

The shared key K for encryption:
        Alice:
ZcSHvJaGe9QVMbMMRFkeO8BIWIXTB1P0V8aCvklqO
        Bobby:
ZcSHvJaGe9QVMbMMRFkeO8BIWIXTB1P0V8aCvklqO

alice:Hi this is alice!
________________________________________
Alice is ENCRYPTING:
Plain Text (17 bytes)
Hi this is alice!

Cipher Text (32 bytes)
0xff 0x9b 0x9f 0xb1 0xc2 0x67 0x30 0x8a 0x22 0x1 0x55 0
   ↪ x3 0xce 0xc5 0x5 0x18 0xfe 0xf 0x45 0xd6 0x62 0x78
   ↪  0x5a 0xe8 0xe 0xb5 0x11 0x39 0x3c 0x65 0xa9 0x3f

________________________________________
________________________________________
Bobby is DECRYPTING:
Cipher Text (20 bytes)
0xff 0x9b 0x9f 0xb1 0xc2 0x67 0x30 0x8a 0x22 0x1 0x55 0
   ↪ x3 0xce 0xc5 0x5 0x18 0xfe 0xf 0x45 0xd6 0x62 0x78
   ↪  0x5a 0xe8 0xe 0xb5 0x11 0x39 0x3c 0x65 0xa9 0x3f

Plain Text (12 bytes)
Hi this is alice!

________________________________________
bobby:Hi alice, this is bobby!
________________________________________
Bobby is ENCRYPTING:
Plain Text (18 bytes)
Hi alice, this is bobby!

Cipher Text (20 bytes)
0xb 0x43 0xee 0xb0 0xdd 0x82 0x8f 0x31 0x1d 0x23 0x29 0
```

```
    ↪ xe 0x49 0x81 0x89 0x99 0xa 0xc1 0xe2 0x7e 0x52 0
    ↪ xe7 0xff 0xd1 0xb5 0xe9 0xca 0xd7 0x52 0xcb 0x7a 0
    ↪ xdb

________________________________________________
________________________________________________

Alice is DECRYPTING:
Cipher Text (20 bytes)
0xb 0x43 0xee 0xb0 0xdd 0x82 0x8f 0x31 0x1d 0x23 0x29 0
    ↪ xe 0x49 0x81 0x89 0x99 0xa 0xc1 0xe2 0x7e 0x52 0
    ↪ xe7 0xff 0xd1 0xb5 0xe9 0xca 0xd7 0x52 0xcb 0x7a 0
    ↪ xdb

Plain Text (19 bytes)
Hi alice , this is bobby!

________________________________________________
```

Conclusion

In conclusion, it is very crucial to be carefoul while implementing cryptographic algorithms. While the algorithms in themselves like Blum Blum Sub, or Diffie-Hellman might seem trivial to implement, it is very easy to make basic but dangerous cryptological mistakes. One of many bugs, and thus security holes might come from bad implementation of strong prime number generator. If this is done incorrectly, it is very hard to find out because the building blocks of out protocoll are not dependant on whether or not some cryptological properties hold or not. In the case of strong prime numbers, we might use any number and encryption and decryption will work perfectly, the key is in choosing the right *key*. The key that is cryptologically secure, in this are we stumble upon things like cyclic groups, prime numbers and generators, the properties of those numbers and mathematical proofs provide us with very difficult to guess keys, sometimes undistinguishable from noise, those combined with different encryption and decryption algorithms can provide secure building blocks for implementing secure system solutions of any kind.

Works Cited

[1] GMP GNU Library for arbitrary precision arithmetic
`https://gmplig.org/`

[2] Quadgram analysis, Practical Cryptograpy.
`https://www.cryptopp.com/`

[3] Fermat's little theorem, primality test
`https://en.wikipedia.org/wiki/Fermat_primality_test`

[4] Integer and prime factorization problem and difficulty.
`https://en.wikipedia.org/wiki/Integer_factorization`

[5] Blum Blum Shub CSPRNG, Wikipedia
`https://en.wikipedia.org/wiki/Blum_Blum_Shub`