PDM with augmentation

Fabio Massimo Ercoli [*]

December 7th 2024

1 Basic concepts

1.1 EKE

EXE (Encrypted Key Exchange) is used to implement mutual authentication between two parties, usually a client and a server. The process requires a shared weak secret key (W) that is derived from the user password. This weak key can be computed by the client when the user inserts the password, so we don't need to store anything on the client machine. While on the server side we need to store the value of W in persistent storage.

The weak key is used to encrypt a Diffie-Hellman key exchange that is used to produce a stronger password. The stronger password is used at the end of the procedure to mutually authenticate the client and the server, using challenges. The authentication based on the strong key so is more robust than the one based on W.

1.2 PDM

PDM (Password Derived Moduli) is a variant of the EKE, in particular it is a simplification where the value p, that is the number we use for the *mod* operation of the Diffie-Hellman key exchange, is no longer a fixed number but it is a number that depends on the password. This allow to choose a fixed value for the generation number g, in case of PDM this value is equals to *2*.

The benefit of this variant is that the two parties only need to share their encrypted public keys values: $W\{2^a mod\, p\}$ and $W\{2^a mod\, p\}$, since g is always 2 and p is derived from the password. Again W is stored on the server machine and derived from the password on the client machine.

1.3 PDM with augmentation

Sniffing the messages exchanged during the *PDM* (or *EKE*) procedure does not allow to find W using an offline dictionary attack, since the weak key is used only

[*]More on me: https://github.com/fax4ever https://www.linkedin.com/in/fabioercoli/.

to encrypt the Diffie-Hellman key exchange. But if the server is compromised, the offline dictionary attack will be applicable to the weak keys stored in the secondary storage of the server and if the attacker knows W will be able to impersonate the user.

So we need to find a way to solve this potential vulnerability to make the authentication even stronger. The idea of *PDM with augmentation* or in general of *EKE with augmentation* is to store a value on the server machine that cannot be used to obtain the weak key W.

In *PDM with augmentation*, as usual we don't need to store anything client side, since the values W and p can be directly derived from the user's password. On the server side, for each registered user, we store the triple:

- The username of the user, for instance *Alice*
- The *mod* operation p
- $2^W mod\, p$

If the value $2^W mod\, p$ is computerized, this can be used only to impersonate the server and not the user. Because to impersonate the user we need to solve the discrete logarithm problem in order to get back to W.
The procedure is performed using just 3 message exchanges:

1. **A** $\rightarrow$ **B**: $2^a mod\, p$
2. **B** $\rightarrow$ **A**: $2^b mod\, p$, *hash1(sessionKey)*
3. **A** $\rightarrow$ **B**: *hash2(sessionKey)*

Where:

- a is the private value chosen by A
- b is the private value chosen by B
- *hash1* and *hash1* are two different cryptographic hashing functions

And:

$$sessionKey = 2^{a \cdot b} mod\, p\ , 2^{b \cdot W} mod\, p \tag{1}$$

The client is able to reconstruct the session key, since it knows a, $2^b mod\, p$ and W. The server is able to reconstruct the session key, since it knows b, $2^a mod\, p$ and $2^W mod\, p$.

2 Implementation

2.1 Calculate the values of the session keys

***p* module value** The first parameter we need to build is p. In general p should be a safe prime, so a prime and also $(p-1)/2$ has to be prime. This is sometimes not considered a strong requirement, since also if p is not a safe prime or even if not prime at all could be considered enough secure choice for p.

We want to generate this value from the user password, possibly salting it with the user name, so that if two users use the same password the value of p is expected to be different. Furthermore, we want to have p large enough to be considered safe against the discrete log computation.

One idea to compute p is to start from the password, salting it with the username, compute the *hash(password* | *username)*, starting from this value we have three options I think:

- Use an odd number close to the value returned from the hash. In this case the value will be a co-prime of the generator 2.
- Find a prime number close to the returned result of the hash.
- Find a safe prime number close to the returned result of the hash.

***a b* private chosen values** Those values can be chosen between *1* and *p-1* using a CSPRGN (cryptographically secure pseudorandom number generators).

***W* derived from the password** This value can be produced using a cryptographic hashing functions taking the password as input and returning a value between *1* and *p-1*.

2.2 Source code of the solution

I implemented a solution based on Bouncy Castle[1] the source code can be found here:Password Derived Moduli project.

The source code is also attached as a zip file named *password-derived-moduli.zip*.

The core of the solution is the class *PDM.java*, the input of the process are: the username (used as salt), the password (used to generate p and W using different hashing functions), and also the size of W and q.

What is q? It is a value I'm using in my implementation to generate p as safe prime from the password. In particular:

$$p = (q \cdot 2) + 1 \tag{2}$$

In my solution I'm using a *PKCS5S2* parameter generator and different cartographic hash functions to derive W and the initial value of q from the password. Then I use the *BigDecimal nextProbablePrime* to compute q and finally p.

In the rest of the code I compute (using the Bouncy Castle Diffie Hellman utilities) the two values:

$$2^{a \cdot b} mod\ p\ ,\ 2^{b \cdot W} mod\ p \tag{3}$$

[1]https://www.bouncycastle.org/

In particular *a* and *b* are chosen uniformly at random in the range [1 - p-1].

3 Conclusions

3.1 Analyze the results

The results are reported in the spreadsheet (file *PDM with augmentation.xlsx*). Open the file we can notice that given a password and a username, keeping the same size and digest algorithm to apply the values of *W*, *initialQ*, *q* and *p* does not change in different executions.

The same is not true for the values *a* and *b* that are chosen differently at any execution.

I tried to use different key sizes.

Also, I tried to apply different hashing functions to derive *W* and *p*. As expected, even if we use the same password, username and sizes, we get different results.

Finally, as expected, changing (even slightly) the password or the username produces results totally different.

3.2 What happens if the server is compromised?

The augmentation, as we said, it is a way to avoid to make more difficult obtain a user password or the value of *W* starting from the values that are stored on the server.

In the case of *PDM with augmentation*, storing $2^W \, mod \, p$ does not allow to get back to *W*, since the discrete log problem is considered to be hard.

The server also needs to store *p* that is derived from the password. Since this value is not a direct result of an hashing function, but different steps are taken in order to compute p, such as derive q and make it prime, maybe we can say we're safe.