RFC 6979 : Deterministic Usage of DSA and ECDSA Digital Signature Algorithms

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 24 août 2013

Date de publication du RFC : Août 2013

https://www.bortzmeyer.org/6979.html

Les algorithmes de signature DSA et ECDSA ont une particularité qui est souvent oubliée : chaque signature doit utiliser un nombre imprévisible. On peut, par exemple, générer ce nombre par un processus aléatoire. Mais tous les systèmes qui voudraient utiliser DSA ou ECDSA n'ont pas forcément un tel générateur aléatoire. Ce RFC propose une autre méthode : fabriquer ce nombre par dérivation **déterministe** d'un certain nombre de paramètres, produisant le même effet, sans pour autant nécessiter de source d'aléa.

Ces deux algorithmes sont très utilisés dans le monde. (À noter qu'ECDSA repose sur les courbes elliptiques, décrites dans le RFC 6090 [1].) Mais cette nécessité de générer un nombre unique et imprévisible **par signature** n'a pas d'équivalent avec des algorithmes comme RSA. Le pourquoi de cette nécessité est bien expliqué dans un article de Nate Lawson <http://rdist.root.org/2010/11/19/dsa-requirements-for-r>. Si vous l'oubliez, vous serez piraté, comme Sony l'a été <http://www.exophase.com/20540/hackers-describe-ps3-security-as-epic-fail-gain-unrestricted-access/>. Mais que ce nombre soit aléatoire n'est pas strictement nécessaire. En fait, l'aléatoire a deux défauts :

— Il rend les tests difficiles puisque deux exécutions successives du même programme ne donneront pas le même résultat. Ces tests ne peuvent pas facilement vérifier si le nombre est bien imprévisible.
— Et, surtout, toutes les machines n'ont pas forcément un générateur aléatoire de bonne qualité (cf. RFC 4086), notamment les engins spécialisés comme les cartes à puce. Cela peut mener à préférer RSA, pourtant plus consommateur de ressources.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6090.txt

L'algorithme permettant de générer un nombre unique et imprévisible, mais sans générateur aléatoire, est décrit dans les sections 2 et 3 de notre RFC (il est fondé sur l'idée de "*derandomization*", décrite dans « "*How Risky is the Random-Oracle Model?*" <http://eprint.iacr.org/2008/441> »). Je ne vais pas résumer cet algorithme ici (la cryptographie, c'est trop fort pour moi) mais ses propriétés importantes sont :

— Les signatures sont des signatures DSA ou ECDSA traditionnelles. Les vérificateurs de signature n'ont pas besoin de connaître ce nouvel algorithme et n'ont donc pas besoin d'être modifiés.
— Les clés, elles, doivent, comme avant, être générées aléatoirement (comme avec RSA).

L'annexe A contient un exemple détaillé de calcul du nombre unique pour une signature.

Si vous envisagez de programmer cet algorithme de génération déterministe de signature, faites bien attention à la section 4, qui résume les problèmes de sécurité. Par exemple, comme rappelé plus haut, la procédure de ce RFC ne s'applique **pas** aux clés qui doivent, comme avant, être générées aléatoirement. Pour des engins sans générateur aléatoire, comme les carte à puce de bas et de milieu de gamme, cela peut par exemple être fait à l'usine, pendant la fabrication de l'engin.

D'autre part, comme la génération de la signature est déterministe, le même messsage aura donc toujours la même signature (tant qu'on garde la même clé privée, bien sûr). Cela n'est pas un problème dans certaines applications (RSA, tel que décrit dans le RFC 3447, fait pareil). Ainsi, TLS (RFC 5246), SSH (RFC 4251) ou CMS (RFC 5652) peuvent utiliser cet algorithme déterministe sans risque pour leur sécurité.

À noter que, de manière surprenante pour un RFC, la section 5 est consacrée aux risque de brevets sur cet algorithme (apparemment assez faibles). C'est surprenant car, comme la situation change tout le temps (nouveaux brevets, brevets reconnus comme futiles, etc), cette information est normalement distribuée en ligne <http://www.ietf.org/ipr/> et pas mise dans un RFC stable (c'est expliqué dans le RFC 8179). Les brevets actuellement connus pour ce RFC sont sur le site Web de l'IETF <https://datatracker.ietf.org/ipr/search/?option=rfc_search&rfc_search=6979>.

L'annexe B contient une mise en œuvre, en Java, de cet algorithme (copiée ici dans le fichier (en ligne sur https://www.bortzmeyer.org/files/DeterministicDSA.java)). On peut l'utiliser, par exemple, ainsi :

```
import java.math.BigInteger;
import org.ietf.DeterministicDSA;

public class TestDeterministicDSA {

    // http://stackoverflow.com/questions/5470219/java-get-md5-string-from-message-digest
    private static String binary2string(byte []bin) {
StringBuilder binString = new StringBuilder();
for (int i = 0; i < bin.length; i++) {
    String hex = Integer.toHexString(bin[i]);
    if (hex.length() == 1)
{
    binString.append('0');
    binString.append(hex.charAt(hex.length() - 1));
}
    else
binString.append(hex.substring(hex.length() - 2));
}
return binString.toString();
    }

    public static void main (String args[]) {
int i;
```

```
        DeterministicDSA dsa = new DeterministicDSA("SHA-256");
String signature;
        // We use the private key for the appendix A.2.1 of the RFC
BigInteger p = new BigInteger("86F5CA03DCFEB225063FF830A0C769B9DD9D6153AD91D7CE27F787C43278B447E6533B86B18BED6E8
BigInteger q = new BigInteger("996F967F6C8E388D9E28D01E205FBA957A5698B1", 16);
BigInteger g = new BigInteger("07B0F92546150B62514BB771E2A0C0CE387F03BDA6C56B505209FF25FD3C133D89BBCD97E904E0911
BigInteger x = new BigInteger("411602CB19A6CCC34494D79D98EF1E7ED5AF25F7", 16);
dsa.setPrivateKey(p, q, g, x);
for (i=0; i<args.length; i++) {
    dsa.update(args[i].getBytes());
    signature = binary2string(dsa.sign());
    System.out.println(args[i] + " -> " + signature);
    dsa.reset();
}
    }

}
```

Et on retrouve les valeurs indiquées dans le RFC (ce qui est logique pour un processus déterministe...) :

```
% java TestDeterministicDSA sample test
sample -> 302d02150081f2f5850be5bc123c43f71a3033e9384611c54502144cdd914b65eb6c66a8aaad27299bee6b035f5e89
test -> 302c021422518c127299b0f6fdc9872b282b9e70d079081202146837ec18f150d55de95b5e29be7af5d01e4fe160
```